目录

**[一、 SPA开发打包生成的bundle太大如何解决](#_Toc17720_WPSOffice_Level1)** **[9](#_Toc17720_WPSOffice_Level1)**

**[1. 通常解法](#_Toc12326_WPSOffice_Level1)** **[9](#_Toc12326_WPSOffice_Level1)**

**[2. 通用大牛级解法](#_Toc14865_WPSOffice_Level1)** **[10](#_Toc14865_WPSOffice_Level1)**

**[3. 解法对比及优缺点](#_Toc22471_WPSOffice_Level1)** **[10](#_Toc22471_WPSOffice_Level1)**

**[4. 延伸及扩展问题回答参考](#_Toc7283_WPSOffice_Level1)** **[10](#_Toc7283_WPSOffice_Level1)**

**[5. 项目中体现经验的点](#_Toc17526_WPSOffice_Level1)** **[10](#_Toc17526_WPSOffice_Level1)**

**[6. 论坛参考](#_Toc15364_WPSOffice_Level1)** **[11](#_Toc15364_WPSOffice_Level1)**

**[二、 说说React生命周期？](#_Toc20872_WPSOffice_Level1)** **[11](#_Toc20872_WPSOffice_Level1)**

[1. 通常解法](#_Toc14865_WPSOffice_Level2) [11](#_Toc14865_WPSOffice_Level2)

[2. 通用大牛级解法](#_Toc22471_WPSOffice_Level2) [11](#_Toc22471_WPSOffice_Level2)

[3. 解法对比及优缺点](#_Toc7283_WPSOffice_Level2) [12](#_Toc7283_WPSOffice_Level2)

[4. 延伸及扩展问题回答参考](#_Toc17526_WPSOffice_Level2) [12](#_Toc17526_WPSOffice_Level2)

[5. 项目中体现经验的点](#_Toc15364_WPSOffice_Level2) [13](#_Toc15364_WPSOffice_Level2)

[6. 论坛参考](#_Toc20872_WPSOffice_Level2) [13](#_Toc20872_WPSOffice_Level2)

**[三、 如何实现发布订阅模式](#_Toc31651_WPSOffice_Level1)** **[0](#_Toc31651_WPSOffice_Level1)**

[1. 通常解法](#_Toc31651_WPSOffice_Level2) [0](#_Toc31651_WPSOffice_Level2)

[2. 通用大牛级解法](#_Toc24526_WPSOffice_Level2) [0](#_Toc24526_WPSOffice_Level2)

[3. 解法对比及优缺点](#_Toc17223_WPSOffice_Level2) [1](#_Toc17223_WPSOffice_Level2)

[4. 延伸及扩展问题回答参考](#_Toc9172_WPSOffice_Level2) [2](#_Toc9172_WPSOffice_Level2)

[5. 项目中体现经验的点](#_Toc8654_WPSOffice_Level2) [2](#_Toc8654_WPSOffice_Level2)

**[四、 封装一个全局的模态框](#_Toc24526_WPSOffice_Level1)** **[0](#_Toc24526_WPSOffice_Level1)**

[1. 通常解法](#_Toc22062_WPSOffice_Level2) [0](#_Toc22062_WPSOffice_Level2)

[2. 通用大牛级解法](#_Toc18075_WPSOffice_Level2) [0](#_Toc18075_WPSOffice_Level2)

[3. 解法对比及优缺点](#_Toc17242_WPSOffice_Level2) [1](#_Toc17242_WPSOffice_Level2)

[4. 延伸及扩展问题回答参考](#_Toc16121_WPSOffice_Level2) [1](#_Toc16121_WPSOffice_Level2)

[5.项目中体现经验的点](#_Toc8884_WPSOffice_Level2) [2](#_Toc8884_WPSOffice_Level2)

[6.论坛参考](#_Toc3919_WPSOffice_Level2) [2](#_Toc3919_WPSOffice_Level2)

**[五、 如何解决vue组件之间的通讯?](#_Toc25730_WPSOffice_Level1)** **[0](#_Toc25730_WPSOffice_Level1)**

[1. 通常解法](#_Toc1465_WPSOffice_Level2) [0](#_Toc1465_WPSOffice_Level2)

[2. 通用大牛级解法](#_Toc30792_WPSOffice_Level2) [1](#_Toc30792_WPSOffice_Level2)

[3. 解法对比及优缺点](#_Toc29668_WPSOffice_Level2) [2](#_Toc29668_WPSOffice_Level2)

[4. 延伸及扩展问题回答参考](#_Toc18727_WPSOffice_Level2) [2](#_Toc18727_WPSOffice_Level2)

[5. 项目中体现经验的点](#_Toc27461_WPSOffice_Level2) [3](#_Toc27461_WPSOffice_Level2)

[6. 论坛参考](#_Toc28843_WPSOffice_Level2) [3](#_Toc28843_WPSOffice_Level2)

**[六、 在主流的vue或react项目中跟后端的数据交互问题?](#_Toc30904_WPSOffice_Level1)** **[3](#_Toc30904_WPSOffice_Level1)**

[1. 通常解法](#_Toc26859_WPSOffice_Level2) [3](#_Toc26859_WPSOffice_Level2)

[2. 通用大牛级解法](#_Toc17347_WPSOffice_Level2) [3](#_Toc17347_WPSOffice_Level2)

[3. 解法对比及优缺点](#_Toc25651_WPSOffice_Level2) [4](#_Toc25651_WPSOffice_Level2)

[4. 延伸及扩展问题回答参考](#_Toc21168_WPSOffice_Level2) [4](#_Toc21168_WPSOffice_Level2)

[5. 项目中体现经验的点](#_Toc12606_WPSOffice_Level2) [4](#_Toc12606_WPSOffice_Level2)

[6. 论坛参考](#_Toc24880_WPSOffice_Level2) [4](#_Toc24880_WPSOffice_Level2)

**[七、 你是如何解决前端开发数据作用域问题?](#_Toc29521_WPSOffice_Level1)** **[4](#_Toc29521_WPSOffice_Level1)**

[1. 通常解法](#_Toc20853_WPSOffice_Level2) [4](#_Toc20853_WPSOffice_Level2)

[2. 通用大牛级解法](#_Toc6018_WPSOffice_Level2) [4](#_Toc6018_WPSOffice_Level2)

[3. 延伸及扩展问题回答参](#_Toc31544_WPSOffice_Level2) [5](#_Toc31544_WPSOffice_Level2)

[4. 项目中体现经验的点](#_Toc32563_WPSOffice_Level2) [6](#_Toc32563_WPSOffice_Level2)

[5. 论坛参考](#_Toc13281_WPSOffice_Level2) [6](#_Toc13281_WPSOffice_Level2)

**[八、 什么是同源策略](#_Toc17223_WPSOffice_Level1)** **[6](#_Toc17223_WPSOffice_Level1)**

[1. 通常解法](#_Toc10791_WPSOffice_Level2) [6](#_Toc10791_WPSOffice_Level2)

[2. 通用大牛级解法](#_Toc18388_WPSOffice_Level2) [7](#_Toc18388_WPSOffice_Level2)

[3. 解法对比及优缺点](#_Toc29955_WPSOffice_Level2) [9](#_Toc29955_WPSOffice_Level2)

[4. 延伸及扩展问题回答参考](#_Toc965_WPSOffice_Level2) [9](#_Toc965_WPSOffice_Level2)

[5. 项目中体现经验的点](#_Toc22977_WPSOffice_Level2) [9](#_Toc22977_WPSOffice_Level2)

[6. 论坛参考](#_Toc16759_WPSOffice_Level2) [9](#_Toc16759_WPSOffice_Level2)

**[九、 Generator async promise 的区别?](#_Toc9172_WPSOffice_Level1)** **[10](#_Toc9172_WPSOffice_Level1)**

[1. 通常解法](#_Toc15362_WPSOffice_Level2) [10](#_Toc15362_WPSOffice_Level2)

[2. 通用大牛级解法](#_Toc11084_WPSOffice_Level2) [10](#_Toc11084_WPSOffice_Level2)

[3. 解法对比及优缺点](#_Toc9437_WPSOffice_Level2) [13](#_Toc9437_WPSOffice_Level2)

[4. 延伸及扩展问题回答参考](#_Toc30030_WPSOffice_Level2) [13](#_Toc30030_WPSOffice_Level2)

[5. 项目中体现经验的点](#_Toc21052_WPSOffice_Level2) [13](#_Toc21052_WPSOffice_Level2)

[6. 论坛参考](#_Toc32676_WPSOffice_Level2) [13](#_Toc32676_WPSOffice_Level2)

**[十、 说一下前端存储?](#_Toc8654_WPSOffice_Level1)** **[13](#_Toc8654_WPSOffice_Level1)**

[1. 通常解法](#_Toc24068_WPSOffice_Level2) [13](#_Toc24068_WPSOffice_Level2)

[2. 通用大牛级解法](#_Toc13711_WPSOffice_Level2) [14](#_Toc13711_WPSOffice_Level2)

[3. 解法对比及优缺点](#_Toc9115_WPSOffice_Level2) [15](#_Toc9115_WPSOffice_Level2)

[4. 延伸及扩展问题回答参考](#_Toc21855_WPSOffice_Level2) [15](#_Toc21855_WPSOffice_Level2)

[5. 项目中体现经验的点](#_Toc32396_WPSOffice_Level2) [15](#_Toc32396_WPSOffice_Level2)

[6. 论坛参考](#_Toc28930_WPSOffice_Level2) [15](#_Toc28930_WPSOffice_Level2)

**[十一、 解释一下javaScript中的垃圾回收机制？](#_Toc22062_WPSOffice_Level1)** **[15](#_Toc22062_WPSOffice_Level1)**

[1. 通常解法](#_Toc4954_WPSOffice_Level2) [16](#_Toc4954_WPSOffice_Level2)

[2. 通用大牛级解法](#_Toc13338_WPSOffice_Level2) [16](#_Toc13338_WPSOffice_Level2)

[3. 解法对比及优缺点](#_Toc7092_WPSOffice_Level2) [17](#_Toc7092_WPSOffice_Level2)

[4. 延伸及扩展问题回答参考](#_Toc9153_WPSOffice_Level2) [17](#_Toc9153_WPSOffice_Level2)

[5. 项目中体现经验的点](#_Toc14779_WPSOffice_Level2) [18](#_Toc14779_WPSOffice_Level2)

[6. 论坛参考](#_Toc27997_WPSOffice_Level2) [18](#_Toc27997_WPSOffice_Level2)

**[十二、 解释一下React中的JSX语法？](#_Toc18075_WPSOffice_Level1)** **[18](#_Toc18075_WPSOffice_Level1)**

[1. 通常解法](#_Toc12785_WPSOffice_Level2) [18](#_Toc12785_WPSOffice_Level2)

[2. 通用大牛级解法](#_Toc17724_WPSOffice_Level2) [19](#_Toc17724_WPSOffice_Level2)

[3. 解法对比及优缺点](#_Toc10749_WPSOffice_Level2) [19](#_Toc10749_WPSOffice_Level2)

[4. 延伸及扩展问题回答参考](#_Toc6851_WPSOffice_Level2) [20](#_Toc6851_WPSOffice_Level2)

[5. 项目中体现经验的点](#_Toc14978_WPSOffice_Level2) [20](#_Toc14978_WPSOffice_Level2)

[6. 论坛参考](#_Toc22399_WPSOffice_Level2) [20](#_Toc22399_WPSOffice_Level2)

**[十三、 redux三大原则？](#_Toc17242_WPSOffice_Level1)** **[20](#_Toc17242_WPSOffice_Level1)**

[1. 单一数据源](#_Toc21640_WPSOffice_Level2) [20](#_Toc21640_WPSOffice_Level2)

[2. State 是只读的](#_Toc18267_WPSOffice_Level2) [21](#_Toc18267_WPSOffice_Level2)

[3. 使用纯函数来执行修改](#_Toc16202_WPSOffice_Level2) [21](#_Toc16202_WPSOffice_Level2)

[4. 延伸及扩展问题回答参考](#_Toc31672_WPSOffice_Level2) [23](#_Toc31672_WPSOffice_Level2)

[5. 项目中体现经验的点](#_Toc22892_WPSOffice_Level2) [23](#_Toc22892_WPSOffice_Level2)

[6. 论坛参考](#_Toc18509_WPSOffice_Level2) [23](#_Toc18509_WPSOffice_Level2)

**[十四、 redux-saga中间件？](#_Toc16121_WPSOffice_Level1)** **[0](#_Toc16121_WPSOffice_Level1)**

[1. 通常解法](#_Toc24799_WPSOffice_Level2) [0](#_Toc24799_WPSOffice_Level2)

[2. 通用大牛级解法](#_Toc12275_WPSOffice_Level2) [0](#_Toc12275_WPSOffice_Level2)

[3. 解法对比及优缺点](#_Toc24329_WPSOffice_Level2) [1](#_Toc24329_WPSOffice_Level2)

[4. 延伸及扩展问题回答参考](#_Toc2241_WPSOffice_Level2) [1](#_Toc2241_WPSOffice_Level2)

[5. 项目中体现经验的点](#_Toc735_WPSOffice_Level2) [1](#_Toc735_WPSOffice_Level2)

[6. 论坛参考](#_Toc6697_WPSOffice_Level2) [2](#_Toc6697_WPSOffice_Level2)

**[十五、 react-redux中的容器组件(container)和展示组件(component)？](#_Toc8884_WPSOffice_Level1)** **[2](#_Toc8884_WPSOffice_Level1)**

[1. 组织应用的组件](#_Toc16655_WPSOffice_Level2) [2](#_Toc16655_WPSOffice_Level2)

[2. 连接Store与组件](#_Toc19708_WPSOffice_Level2) [2](#_Toc19708_WPSOffice_Level2)

[3. 通常解法](#_Toc4739_WPSOffice_Level2) [3](#_Toc4739_WPSOffice_Level2)

[4. 延伸及扩展问题回答参考](#_Toc1145_WPSOffice_Level2) [4](#_Toc1145_WPSOffice_Level2)

[5. 项目中体现经验的点](#_Toc11178_WPSOffice_Level2) [4](#_Toc11178_WPSOffice_Level2)

[6. 论坛参考](#_Toc12492_WPSOffice_Level2) [4](#_Toc12492_WPSOffice_Level2)

**[十六、 简述前端性能优化的措施？](#_Toc12076_WPSOffice_Level1)** **[4](#_Toc12076_WPSOffice_Level1)**

[1. 通常解法](#_Toc23075_WPSOffice_Level2) [4](#_Toc23075_WPSOffice_Level2)

[2. 通用大牛级解法](#_Toc32729_WPSOffice_Level2) [5](#_Toc32729_WPSOffice_Level2)

[3. 解法对比及优缺点](#_Toc13766_WPSOffice_Level2) [6](#_Toc13766_WPSOffice_Level2)

[4. 延伸及扩展问题回答参考](#_Toc17598_WPSOffice_Level2) [6](#_Toc17598_WPSOffice_Level2)

[5. 项目中体现经验的点](#_Toc21453_WPSOffice_Level2) [8](#_Toc21453_WPSOffice_Level2)

[6.论坛参考](#_Toc18477_WPSOffice_Level2) [8](#_Toc18477_WPSOffice_Level2)

**[十七、 你是如何解决移动端适配问题？](#_Toc23075_WPSOffice_Level1)** **[8](#_Toc23075_WPSOffice_Level1)**

[1. 通常解法](#_Toc17944_WPSOffice_Level2) [8](#_Toc17944_WPSOffice_Level2)

[2. 通用大牛级解法](#_Toc15395_WPSOffice_Level2) [9](#_Toc15395_WPSOffice_Level2)

[3. 解法对比及优缺点](#_Toc18378_WPSOffice_Level2) [10](#_Toc18378_WPSOffice_Level2)

**[十八、 谈谈对node对的看法？](#_Toc32729_WPSOffice_Level1)** **[11](#_Toc32729_WPSOffice_Level1)**

[1. 通常解法](#_Toc25992_WPSOffice_Level2) [11](#_Toc25992_WPSOffice_Level2)

[2. 通用大牛级解法](#_Toc16112_WPSOffice_Level2) [11](#_Toc16112_WPSOffice_Level2)

[3. 解法对比及优缺点](#_Toc3732_WPSOffice_Level2) [12](#_Toc3732_WPSOffice_Level2)

[4. 延伸及扩展问题回答参考](#_Toc19073_WPSOffice_Level2) [12](#_Toc19073_WPSOffice_Level2)

[5. 。项目中体现经验的点](#_Toc6878_WPSOffice_Level2) [13](#_Toc6878_WPSOffice_Level2)

[6. 论坛参考](#_Toc20299_WPSOffice_Level2) [13](#_Toc20299_WPSOffice_Level2)

**[十九、 Vue的生命周期？](#_Toc13766_WPSOffice_Level1)** **[13](#_Toc13766_WPSOffice_Level1)**

[1. 通常解法](#_Toc27557_WPSOffice_Level2) [13](#_Toc27557_WPSOffice_Level2)

[2. 通用大牛级解法](#_Toc11044_WPSOffice_Level2) [14](#_Toc11044_WPSOffice_Level2)

[3. 解法对比及优缺点](#_Toc10934_WPSOffice_Level2) [16](#_Toc10934_WPSOffice_Level2)

[4. 延伸及扩展问题回答参考](#_Toc16068_WPSOffice_Level2) [16](#_Toc16068_WPSOffice_Level2)

[5. 项目中体现经验的点](#_Toc9083_WPSOffice_Level2) [16](#_Toc9083_WPSOffice_Level2)

[6. 论坛参考](#_Toc18209_WPSOffice_Level2) [16](#_Toc18209_WPSOffice_Level2)

**[二十、 说一下vuex？](#_Toc15395_WPSOffice_Level1)** **[16](#_Toc15395_WPSOffice_Level1)**

[1. 通常解法](#_Toc2435_WPSOffice_Level2) [17](#_Toc2435_WPSOffice_Level2)

[2. 通用大牛级解法](#_Toc4686_WPSOffice_Level2) [17](#_Toc4686_WPSOffice_Level2)

[3. 解法对比及优缺点](#_Toc16460_WPSOffice_Level2) [18](#_Toc16460_WPSOffice_Level2)

[4. 项目中体现经验的点](#_Toc29860_WPSOffice_Level2) [18](#_Toc29860_WPSOffice_Level2)

[5. 论坛参考](#_Toc5716_WPSOffice_Level2) [18](#_Toc5716_WPSOffice_Level2)

**[二十一、 小程序的组件？](#_Toc18378_WPSOffice_Level1)** **[18](#_Toc18378_WPSOffice_Level1)**

[7. 通常解法 8](#_Toc19315_WPSOffice_Level2)

[8. 通用大牛级解法 9](#_Toc3046_WPSOffice_Level2)

[9. 解法对比及优缺点 9](#_Toc19700_WPSOffice_Level2)

[10. 延伸及扩展问题回答参考 10](#_Toc31558_WPSOffice_Level2)

[11. 项目中体现经验的点 10](#_Toc29131_WPSOffice_Level2)

[12. 论坛参考 10](#_Toc18431_WPSOffice_Level2)

**[二十二、 如何理解http的缓存机制？](#_Toc25582_WPSOffice_Level1)** **[20](#_Toc25582_WPSOffice_Level1)**

[1. 通常解法](#_Toc18444_WPSOffice_Level2) [20](#_Toc18444_WPSOffice_Level2)

[2. 通用大牛级解法](#_Toc13756_WPSOffice_Level2) [22](#_Toc13756_WPSOffice_Level2)

[3. 解法对比及优缺点](#_Toc20115_WPSOffice_Level2) [24](#_Toc20115_WPSOffice_Level2)

[4. 延伸及扩展问题回答参考](#_Toc24326_WPSOffice_Level2) [24](#_Toc24326_WPSOffice_Level2)

[5. 项目中体现经验的点](#_Toc23224_WPSOffice_Level2) [24](#_Toc23224_WPSOffice_Level2)

[6.论坛参考](#_Toc26287_WPSOffice_Level2) [25](#_Toc26287_WPSOffice_Level2)

**[二十三、 vue-router中路由组件传参的问题？](#_Toc30948_WPSOffice_Level1)** **[25](#_Toc30948_WPSOffice_Level1)**

[1. 通常解法](#_Toc29459_WPSOffice_Level2) [25](#_Toc29459_WPSOffice_Level2)

[2. 通用大牛级解法](#_Toc29880_WPSOffice_Level2) [26](#_Toc29880_WPSOffice_Level2)

[3. 解法对比及优缺点](#_Toc5189_WPSOffice_Level2) [27](#_Toc5189_WPSOffice_Level2)

[4. 延伸及扩展问题回答参考](#_Toc2468_WPSOffice_Level2) [27](#_Toc2468_WPSOffice_Level2)

[5. 项目中体现经验的点](#_Toc117_WPSOffice_Level2) [27](#_Toc117_WPSOffice_Level2)

[6.论坛参考](#_Toc10332_WPSOffice_Level2) [27](#_Toc10332_WPSOffice_Level2)

**[二十四、 解释一下javaScript中的继承机制？](#_Toc18892_WPSOffice_Level1)** **[0](#_Toc18892_WPSOffice_Level1)**

[1. 通常解法](#_Toc10884_WPSOffice_Level2) [0](#_Toc10884_WPSOffice_Level2)

[2. 通用大牛级解法](#_Toc6561_WPSOffice_Level2) [0](#_Toc6561_WPSOffice_Level2)

[3. 解法对比及优缺点](#_Toc512_WPSOffice_Level2) [2](#_Toc512_WPSOffice_Level2)

[4. 延伸及扩展问题回答参考](#_Toc9324_WPSOffice_Level2) [2](#_Toc9324_WPSOffice_Level2)

[5. 项目中体现经验的点](#_Toc29821_WPSOffice_Level2) [4](#_Toc29821_WPSOffice_Level2)

[6. 论坛参考](#_Toc16513_WPSOffice_Level2) [4](#_Toc16513_WPSOffice_Level2)

**[二十五、 解释一下经常使用的vue数据管理方案？](#_Toc16112_WPSOffice_Level1)** **[0](#_Toc16112_WPSOffice_Level1)**

[1. 通常解法](#_Toc16701_WPSOffice_Level2) [0](#_Toc16701_WPSOffice_Level2)

[2. 通用大牛级解法](#_Toc1118_WPSOffice_Level2) [0](#_Toc1118_WPSOffice_Level2)

[3. 解法对比及优缺点](#_Toc20462_WPSOffice_Level2) [1](#_Toc20462_WPSOffice_Level2)

[4. 延伸及扩展问题回答参考](#_Toc16277_WPSOffice_Level2) [1](#_Toc16277_WPSOffice_Level2)

[5. 项目中体现经验的点](#_Toc1544_WPSOffice_Level2) [2](#_Toc1544_WPSOffice_Level2)

[6. 论坛参考](#_Toc3369_WPSOffice_Level2) [2](#_Toc3369_WPSOffice_Level2)

**[二十六、 Js的运行原理？](#_Toc3732_WPSOffice_Level1)** **[2](#_Toc3732_WPSOffice_Level1)**

[7. 通常解法 8](#_Toc19315_WPSOffice_Level2)

[8. 通用大牛级解法 9](#_Toc3046_WPSOffice_Level2)

[9. 解法对比及优缺点 9](#_Toc19700_WPSOffice_Level2)

[10. 延伸及扩展问题回答参考 10](#_Toc31558_WPSOffice_Level2)

[11. 项目中体现经验的点 10](#_Toc29131_WPSOffice_Level2)

[12. 论坛参考 10](#_Toc18431_WPSOffice_Level2)

**[二十七、 说一下前端安全？](#_Toc19073_WPSOffice_Level1)** **[5](#_Toc19073_WPSOffice_Level1)**

[7. 通常解法 8](#_Toc19315_WPSOffice_Level2)

[8. 通用大牛级解法 9](#_Toc3046_WPSOffice_Level2)

[9. 解法对比及优缺点 9](#_Toc19700_WPSOffice_Level2)

[10. 延伸及扩展问题回答参考 10](#_Toc31558_WPSOffice_Level2)

[11. 项目中体现经验的点 10](#_Toc29131_WPSOffice_Level2)

[12. 论坛参考 10](#_Toc18431_WPSOffice_Level2)

**[二十八、 Promise对象的理解？](#_Toc6878_WPSOffice_Level1)** **[8](#_Toc6878_WPSOffice_Level1)**

[1. 通常解法](#_Toc18497_WPSOffice_Level2) [8](#_Toc18497_WPSOffice_Level2)

[2. 通用大牛级解法](#_Toc30855_WPSOffice_Level2) [9](#_Toc30855_WPSOffice_Level2)

[3. 解法对比及优缺点](#_Toc32251_WPSOffice_Level2) [9](#_Toc32251_WPSOffice_Level2)

[4. 延伸及扩展问题回答参考](#_Toc26451_WPSOffice_Level2) [10](#_Toc26451_WPSOffice_Level2)

[5. 项目中体现经验的点](#_Toc15599_WPSOffice_Level2) [10](#_Toc15599_WPSOffice_Level2)

[6. 论坛参考](#_Toc24098_WPSOffice_Level2) [10](#_Toc24098_WPSOffice_Level2)

**[二十九、 解释一下移动端多倍屏1px像素问题：陈蔓杰](#_Toc20299_WPSOffice_Level1)** **[0](#_Toc20299_WPSOffice_Level1)**

[1. 通常解法](#_Toc17443_WPSOffice_Level2) [0](#_Toc17443_WPSOffice_Level2)

[2. 通用大牛级解法](#_Toc1055_WPSOffice_Level2) [1](#_Toc1055_WPSOffice_Level2)

[3. 解法对比及优缺点](#_Toc17826_WPSOffice_Level2) [2](#_Toc17826_WPSOffice_Level2)

[4. 延伸及扩展问题回答参考](#_Toc5061_WPSOffice_Level2) [2](#_Toc5061_WPSOffice_Level2)

[5. 项目中体现经验的点](#_Toc27913_WPSOffice_Level2) [2](#_Toc27913_WPSOffice_Level2)

[6. 论坛参考](#_Toc25379_WPSOffice_Level2) [2](#_Toc25379_WPSOffice_Level2)

**[三十、 解释一下经常使用的数据类型检测：陈蔓杰](#_Toc1055_WPSOffice_Level1)** **[4](#_Toc1055_WPSOffice_Level1)**

[1. 通常解法](#_Toc21589_WPSOffice_Level2) [4](#_Toc21589_WPSOffice_Level2)

[2. 通用大牛级解法](#_Toc15017_WPSOffice_Level2) [4](#_Toc15017_WPSOffice_Level2)

[3. 解法对比及优缺点](#_Toc31907_WPSOffice_Level2) [6](#_Toc31907_WPSOffice_Level2)

[4. 延伸及扩展问题回答参考](#_Toc12230_WPSOffice_Level2) [6](#_Toc12230_WPSOffice_Level2)

[5. 项目中体现经验的点](#_Toc25434_WPSOffice_Level2) [7](#_Toc25434_WPSOffice_Level2)

[6. 论坛参考](#_Toc30992_WPSOffice_Level2) [7](#_Toc30992_WPSOffice_Level2)

**[三十一、 什么是MVVM？：刘于项目经理](#_Toc17826_WPSOffice_Level1)** **[7](#_Toc17826_WPSOffice_Level1)**

**[1. 通常解法](#_Toc5061_WPSOffice_Level1)** **[7](#_Toc5061_WPSOffice_Level1)**

[2. 解法对比及优缺点](#_Toc8263_WPSOffice_Level2) [8](#_Toc8263_WPSOffice_Level2)

[3. 项目中体现经验的点](#_Toc17548_WPSOffice_Level2) [8](#_Toc17548_WPSOffice_Level2)

[4. 论坛参考](#_Toc20789_WPSOffice_Level2) [8](#_Toc20789_WPSOffice_Level2)

**[三十二、 Vue的响应原理：陈蔓杰项目经理](#_Toc25379_WPSOffice_Level1)** **[8](#_Toc25379_WPSOffice_Level1)**

[1. 通常解法](#_Toc10562_WPSOffice_Level2) [8](#_Toc10562_WPSOffice_Level2)

[2. 通用大牛级解法](#_Toc10162_WPSOffice_Level2) [9](#_Toc10162_WPSOffice_Level2)

[3. 解法对比及优缺点](#_Toc22157_WPSOffice_Level2) [10](#_Toc22157_WPSOffice_Level2)

[4. 项目中体现经验的点](#_Toc20904_WPSOffice_Level2) [10](#_Toc20904_WPSOffice_Level2)

[5. 论坛参考](#_Toc30403_WPSOffice_Level2) [10](#_Toc30403_WPSOffice_Level2)

**[三十三、 高阶组件原理：陈蔓杰项目经理](#_Toc21589_WPSOffice_Level1)** **[10](#_Toc21589_WPSOffice_Level1)**

[1. 通常解法](#_Toc31715_WPSOffice_Level2) [10](#_Toc31715_WPSOffice_Level2)

[2. 通用大牛级解法](#_Toc10270_WPSOffice_Level2) [10](#_Toc10270_WPSOffice_Level2)

[3. 解法对比及优缺点](#_Toc26059_WPSOffice_Level2) [12](#_Toc26059_WPSOffice_Level2)

[4. 项目中体现经验的点](#_Toc17948_WPSOffice_Level2) [12](#_Toc17948_WPSOffice_Level2)

[5. 论坛参考](#_Toc30671_WPSOffice_Level2) [13](#_Toc30671_WPSOffice_Level2)

**[三十四、 React与Vue的区别与联系原理：陈蔓杰项目经理](#_Toc15017_WPSOffice_Level1)** **[13](#_Toc15017_WPSOffice_Level1)**

[1. 通常解法](#_Toc20141_WPSOffice_Level2) [13](#_Toc20141_WPSOffice_Level2)

[2. 通用大牛级解法](#_Toc6750_WPSOffice_Level2) [13](#_Toc6750_WPSOffice_Level2)

[3. 解法对比及优缺点](#_Toc18910_WPSOffice_Level2) [15](#_Toc18910_WPSOffice_Level2)

[4. 项目中体现经验的点](#_Toc14679_WPSOffice_Level2) [15](#_Toc14679_WPSOffice_Level2)

[5. 论坛参考](#_Toc4077_WPSOffice_Level2) [15](#_Toc4077_WPSOffice_Level2)

**[三十五、 解释一下跨域有几种方法？](#_Toc31907_WPSOffice_Level1)** **[15](#_Toc31907_WPSOffice_Level1)**

[7. 通常解法 8](#_Toc19315_WPSOffice_Level2)

[8. 通用大牛级解法 9](#_Toc3046_WPSOffice_Level2)

[9. 解法对比及优缺点 9](#_Toc19700_WPSOffice_Level2)

[10. 延伸及扩展问题回答参考 10](#_Toc31558_WPSOffice_Level2)

[11. 项目中体现经验的点 10](#_Toc29131_WPSOffice_Level2)

[12. 论坛参考 10](#_Toc18431_WPSOffice_Level2)

**[三十六、 解释一下jsonp的原理？](#_Toc8263_WPSOffice_Level1)** **[0](#_Toc8263_WPSOffice_Level1)**

[7. 通常解法](#_Toc93_WPSOffice_Level2) [0](#_Toc93_WPSOffice_Level2)

[8. 通用大牛级解法](#_Toc26853_WPSOffice_Level2) [0](#_Toc26853_WPSOffice_Level2)

[9. 解法对比及优缺点](#_Toc32375_WPSOffice_Level2) [1](#_Toc32375_WPSOffice_Level2)

[10. 延伸及扩展问题回答参考](#_Toc6708_WPSOffice_Level2) [1](#_Toc6708_WPSOffice_Level2)

[11. 项目中体现经验的点](#_Toc5077_WPSOffice_Level2) [2](#_Toc5077_WPSOffice_Level2)

[12. 论坛参考](#_Toc23002_WPSOffice_Level2) [2](#_Toc23002_WPSOffice_Level2)

**[三十七、 Cookie的？](#_Toc17548_WPSOffice_Level1)** **[2](#_Toc17548_WPSOffice_Level1)**

[6. 通常解法](#_Toc25280_WPSOffice_Level2) [3](#_Toc25280_WPSOffice_Level2)

[7. 通用大牛级解法](#_Toc7201_WPSOffice_Level2) [3](#_Toc7201_WPSOffice_Level2)

[8. 解法对比及优缺点](#_Toc5994_WPSOffice_Level2) [5](#_Toc5994_WPSOffice_Level2)

[9. 项目中体现经验的点](#_Toc5407_WPSOffice_Level2) [5](#_Toc5407_WPSOffice_Level2)

[10. 论坛参考](#_Toc15712_WPSOffice_Level2) [5](#_Toc15712_WPSOffice_Level2)

**[三十八、 说一下vue实现双向绑定原理？](#_Toc20789_WPSOffice_Level1)** **[5](#_Toc20789_WPSOffice_Level1)**

[7. 通常解法 8](#_Toc19315_WPSOffice_Level2)

[8. 通用大牛级解法 9](#_Toc3046_WPSOffice_Level2)

[9. 解法对比及优缺点 9](#_Toc19700_WPSOffice_Level2)

[10. 延伸及扩展问题回答参考 10](#_Toc31558_WPSOffice_Level2)

[11. 项目中体现经验的点 10](#_Toc29131_WPSOffice_Level2)

[12. 论坛参考 10](#_Toc18431_WPSOffice_Level2)

**[三十九、 解释一下 XSS与CSRF区别？](#_Toc10562_WPSOffice_Level1)** **[8](#_Toc10562_WPSOffice_Level1)**

[7. 通常解法](#_Toc19315_WPSOffice_Level2) [8](#_Toc19315_WPSOffice_Level2)

[8. 通用大牛级解法](#_Toc3046_WPSOffice_Level2) [9](#_Toc3046_WPSOffice_Level2)

[9. 解法对比及优缺点](#_Toc19700_WPSOffice_Level2) [9](#_Toc19700_WPSOffice_Level2)

[10. 延伸及扩展问题回答参考](#_Toc31558_WPSOffice_Level2) [10](#_Toc31558_WPSOffice_Level2)

[11. 项目中体现经验的点](#_Toc29131_WPSOffice_Level2) [10](#_Toc29131_WPSOffice_Level2)

[12. 论坛参考](#_Toc18431_WPSOffice_Level2) [10](#_Toc18431_WPSOffice_Level2)

**[四十、 解释一下函数节流？](#_Toc11804_WPSOffice_Level1)** **[0](#_Toc11804_WPSOffice_Level1)**

[7. 通常解法](#_Toc11804_WPSOffice_Level2) [0](#_Toc11804_WPSOffice_Level2)

[8. 通用大牛级解法](#_Toc2922_WPSOffice_Level2) [0](#_Toc2922_WPSOffice_Level2)

[9. 解法对比及优缺点](#_Toc29125_WPSOffice_Level2) [1](#_Toc29125_WPSOffice_Level2)

[10. 延伸及扩展问题回答参考](#_Toc5459_WPSOffice_Level2) [1](#_Toc5459_WPSOffice_Level2)

[11. 项目中体现经验的点](#_Toc29306_WPSOffice_Level2) [1](#_Toc29306_WPSOffice_Level2)

[12. 论坛参考](#_Toc24702_WPSOffice_Level2) [1](#_Toc24702_WPSOffice_Level2)

**[四十一、 解释事件“捕获”，“冒泡”执行原理？](#_Toc2922_WPSOffice_Level1)** **[2](#_Toc2922_WPSOffice_Level1)**

[19. 通常解法 35](#_Toc31030_WPSOffice_Level2)

[20. 通用大牛级解法 35](#_Toc26240_WPSOffice_Level2)

[21. 解法对比及优缺点 38](#_Toc24565_WPSOffice_Level2)

[22. 延伸及扩展问题回答参考 38](#_Toc11916_WPSOffice_Level2)

[23. 项目中体现经验的点 38](#_Toc21778_WPSOffice_Level2)

[24. 论坛参考 38](#_Toc8464_WPSOffice_Level2)

**[四十二、 解释一下页面重构操作？](#_Toc29125_WPSOffice_Level1)** **[5](#_Toc29125_WPSOffice_Level1)**

[7. 通常解法](#_Toc12090_WPSOffice_Level2) [5](#_Toc12090_WPSOffice_Level2)

[8. 通用大牛级解法](#_Toc15650_WPSOffice_Level2) [5](#_Toc15650_WPSOffice_Level2)

[9. 解法对比及优缺点](#_Toc6082_WPSOffice_Level2) [6](#_Toc6082_WPSOffice_Level2)

[10. 延伸及扩展问题回答参考](#_Toc25287_WPSOffice_Level2) [6](#_Toc25287_WPSOffice_Level2)

[11. 项目中体现经验的点](#_Toc31036_WPSOffice_Level2) [6](#_Toc31036_WPSOffice_Level2)

[12. 论坛参考](#_Toc30402_WPSOffice_Level2) [6](#_Toc30402_WPSOffice_Level2)

**[四十四、 谈谈对fetch对的看法？](#_Toc5459_WPSOffice_Level1)** **[7](#_Toc5459_WPSOffice_Level1)**

[7. 通常解法](#_Toc1326_WPSOffice_Level2) [7](#_Toc1326_WPSOffice_Level2)

[8. 通用大牛级解法](#_Toc10213_WPSOffice_Level2) [7](#_Toc10213_WPSOffice_Level2)

[9. 解法对比及优缺点](#_Toc27900_WPSOffice_Level2) [8](#_Toc27900_WPSOffice_Level2)

[10. 延伸及扩展问题回答参考](#_Toc4212_WPSOffice_Level2) [8](#_Toc4212_WPSOffice_Level2)

[11. 。项目中体现经验的点](#_Toc7814_WPSOffice_Level2) [8](#_Toc7814_WPSOffice_Level2)

[12. 论坛参考](#_Toc22734_WPSOffice_Level2) [8](#_Toc22734_WPSOffice_Level2)

**[四十五、 解释一下同源策略？](#_Toc29306_WPSOffice_Level1)** **[9](#_Toc29306_WPSOffice_Level1)**

[7. 通常解法](#_Toc21159_WPSOffice_Level2) [9](#_Toc21159_WPSOffice_Level2)

[8. 通用大牛级解法](#_Toc10362_WPSOffice_Level2) [9](#_Toc10362_WPSOffice_Level2)

[9. 解法对比及优缺点](#_Toc25751_WPSOffice_Level2) [11](#_Toc25751_WPSOffice_Level2)

[10. 延伸及扩展问题回答参考](#_Toc8305_WPSOffice_Level2) [11](#_Toc8305_WPSOffice_Level2)

[11. 项目中体现经验的点](#_Toc24963_WPSOffice_Level2) [12](#_Toc24963_WPSOffice_Level2)

[12. 论坛参考](#_Toc12506_WPSOffice_Level2) [12](#_Toc12506_WPSOffice_Level2)

**[四十六、 说一下jsonp工作原理？](#_Toc25287_WPSOffice_Level1)** **[12](#_Toc25287_WPSOffice_Level1)**

[19. 通常解法 35](#_Toc31030_WPSOffice_Level2)

[20. 通用大牛级解法 35](#_Toc26240_WPSOffice_Level2)

[21. 解法对比及优缺点 38](#_Toc24565_WPSOffice_Level2)

[22. 延伸及扩展问题回答参考 38](#_Toc11916_WPSOffice_Level2)

[23. 项目中体现经验的点 38](#_Toc21778_WPSOffice_Level2)

[24. 论坛参考 38](#_Toc8464_WPSOffice_Level2)

**[四十七、 JavaScript作用域链？](#_Toc31036_WPSOffice_Level1)** **[13](#_Toc31036_WPSOffice_Level1)**

[19. 通常解法 35](#_Toc31030_WPSOffice_Level2)

[20. 通用大牛级解法 35](#_Toc26240_WPSOffice_Level2)

[21. 解法对比及优缺点 38](#_Toc24565_WPSOffice_Level2)

[22. 延伸及扩展问题回答参考 38](#_Toc11916_WPSOffice_Level2)

[23. 项目中体现经验的点 38](#_Toc21778_WPSOffice_Level2)

[24. 论坛参考 38](#_Toc8464_WPSOffice_Level2)

**[四十八、 谈谈RESTful？](#_Toc30402_WPSOffice_Level1)** **[15](#_Toc30402_WPSOffice_Level1)**

[7. 组织应用的组件](#_Toc31200_WPSOffice_Level2) [16](#_Toc31200_WPSOffice_Level2)

[8. 连接Store与组件](#_Toc28058_WPSOffice_Level2) [16](#_Toc28058_WPSOffice_Level2)

[9. 通常解法](#_Toc2618_WPSOffice_Level2) [16](#_Toc2618_WPSOffice_Level2)

[10. 延伸及扩展问题回答参考](#_Toc24672_WPSOffice_Level2) [17](#_Toc24672_WPSOffice_Level2)

[11. 项目中体现经验的点](#_Toc24462_WPSOffice_Level2) [17](#_Toc24462_WPSOffice_Level2)

[12. 论坛参考](#_Toc3094_WPSOffice_Level2) [17](#_Toc3094_WPSOffice_Level2)

**[四十九、 简述cookie？](#_Toc1326_WPSOffice_Level1)** **[17](#_Toc1326_WPSOffice_Level1)**

[6. 通常解法](#_Toc15222_WPSOffice_Level2) [18](#_Toc15222_WPSOffice_Level2)

[7. 通用大牛级解法](#_Toc13538_WPSOffice_Level2) [19](#_Toc13538_WPSOffice_Level2)

[8. 解法对比及优缺点](#_Toc32427_WPSOffice_Level2) [20](#_Toc32427_WPSOffice_Level2)

[9. 延伸及扩展问题回答参考](#_Toc22807_WPSOffice_Level2) [20](#_Toc22807_WPSOffice_Level2)

[10. 项目中体现经验的点](#_Toc21677_WPSOffice_Level2) [21](#_Toc21677_WPSOffice_Level2)

[6.论坛参考](#_Toc27335_WPSOffice_Level2) [21](#_Toc27335_WPSOffice_Level2)

**[五十、 谈谈vuejs变化检测问题？](#_Toc10213_WPSOffice_Level1)** **[21](#_Toc10213_WPSOffice_Level1)**

[19. 通常解法 35](#_Toc31030_WPSOffice_Level2)

[20. 通用大牛级解法 35](#_Toc26240_WPSOffice_Level2)

[21. 解法对比及优缺点 38](#_Toc24565_WPSOffice_Level2)

[22. 延伸及扩展问题回答参考 38](#_Toc11916_WPSOffice_Level2)

[23. 项目中体现经验的点 38](#_Toc21778_WPSOffice_Level2)

[24. 论坛参考 38](#_Toc8464_WPSOffice_Level2)

**[五十一、 谈谈对nodejs对的看法？](#_Toc27900_WPSOffice_Level1)** **[24](#_Toc27900_WPSOffice_Level1)**

[13. 通常解法](#_Toc17854_WPSOffice_Level2) [24](#_Toc17854_WPSOffice_Level2)

[14. 通用大牛级解法](#_Toc1788_WPSOffice_Level2) [25](#_Toc1788_WPSOffice_Level2)

[15. 解法对比及优缺点](#_Toc3395_WPSOffice_Level2) [25](#_Toc3395_WPSOffice_Level2)

[16. 延伸及扩展问题回答参考](#_Toc22552_WPSOffice_Level2) [26](#_Toc22552_WPSOffice_Level2)

[17. 。项目中体现经验的点](#_Toc20786_WPSOffice_Level2) [26](#_Toc20786_WPSOffice_Level2)

[18. 论坛参考](#_Toc20420_WPSOffice_Level2) [26](#_Toc20420_WPSOffice_Level2)

**[五十二、 谈谈Vuejs？](#_Toc22552_WPSOffice_Level1)** **[26](#_Toc22552_WPSOffice_Level1)**

[13. 通常解法](#_Toc15774_WPSOffice_Level2) [26](#_Toc15774_WPSOffice_Level2)

[14. 通用大牛级解法](#_Toc465_WPSOffice_Level2) [27](#_Toc465_WPSOffice_Level2)

[15. 解法对比及优缺点](#_Toc27814_WPSOffice_Level2) [29](#_Toc27814_WPSOffice_Level2)

[16. 延伸及扩展问题回答参考](#_Toc9162_WPSOffice_Level2) [29](#_Toc9162_WPSOffice_Level2)

[17. 项目中体现经验的点](#_Toc32162_WPSOffice_Level2) [29](#_Toc32162_WPSOffice_Level2)

[18. 论坛参考](#_Toc32702_WPSOffice_Level2) [29](#_Toc32702_WPSOffice_Level2)

**[五十三、 说一下keep-alive？](#_Toc27814_WPSOffice_Level1)** **[29](#_Toc27814_WPSOffice_Level1)**

[19. 通常解法 35](#_Toc31030_WPSOffice_Level2)

[20. 通用大牛级解法 35](#_Toc26240_WPSOffice_Level2)

[21. 解法对比及优缺点 38](#_Toc24565_WPSOffice_Level2)

[22. 延伸及扩展问题回答参考 38](#_Toc11916_WPSOffice_Level2)

[23. 项目中体现经验的点 38](#_Toc21778_WPSOffice_Level2)

[24. 论坛参考 38](#_Toc8464_WPSOffice_Level2)

**[五十四、 谈谈vue脚手架？](#_Toc9162_WPSOffice_Level1)** **[31](#_Toc9162_WPSOffice_Level1)**

[19. 通常解法 35](#_Toc31030_WPSOffice_Level2)

[20. 通用大牛级解法 35](#_Toc26240_WPSOffice_Level2)

[21. 解法对比及优缺点 38](#_Toc24565_WPSOffice_Level2)

[22. 延伸及扩展问题回答参考 38](#_Toc11916_WPSOffice_Level2)

[23. 项目中体现经验的点 38](#_Toc21778_WPSOffice_Level2)

[24. 论坛参考 38](#_Toc8464_WPSOffice_Level2)

**[五十五、 谈谈对node对的看法？](#_Toc32162_WPSOffice_Level1)** **[33](#_Toc32162_WPSOffice_Level1)**

[19. 通常解法](#_Toc25139_WPSOffice_Level2) [33](#_Toc25139_WPSOffice_Level2)

[20. 通用大牛级解法](#_Toc31080_WPSOffice_Level2) [33](#_Toc31080_WPSOffice_Level2)

[21. 解法对比及优缺点](#_Toc19772_WPSOffice_Level2) [34](#_Toc19772_WPSOffice_Level2)

[22. 延伸及扩展问题回答参考](#_Toc17872_WPSOffice_Level2) [34](#_Toc17872_WPSOffice_Level2)

[23. 。项目中体现经验的点](#_Toc32405_WPSOffice_Level2) [35](#_Toc32405_WPSOffice_Level2)

[24. 论坛参考](#_Toc12470_WPSOffice_Level2) [35](#_Toc12470_WPSOffice_Level2)

**[五十六、 Vue的脚手架？](#_Toc32702_WPSOffice_Level1)** **[35](#_Toc32702_WPSOffice_Level1)**

[19. 通常解法](#_Toc31030_WPSOffice_Level2) [35](#_Toc31030_WPSOffice_Level2)

[20. 通用大牛级解法](#_Toc26240_WPSOffice_Level2) [35](#_Toc26240_WPSOffice_Level2)

[21. 解法对比及优缺点](#_Toc24565_WPSOffice_Level2) [38](#_Toc24565_WPSOffice_Level2)

[22. 延伸及扩展问题回答参考](#_Toc11916_WPSOffice_Level2) [38](#_Toc11916_WPSOffice_Level2)

[23. 项目中体现经验的点](#_Toc21778_WPSOffice_Level2) [38](#_Toc21778_WPSOffice_Level2)

[24. 论坛参考](#_Toc8464_WPSOffice_Level2) [38](#_Toc8464_WPSOffice_Level2)

**[五十七、 说一下seo？](#_Toc32405_WPSOffice_Level1)** **[38](#_Toc32405_WPSOffice_Level1)**

[19. 通常解法 35](#_Toc31030_WPSOffice_Level2)

[20. 通用大牛级解法 35](#_Toc26240_WPSOffice_Level2)

[21. 解法对比及优缺点 38](#_Toc24565_WPSOffice_Level2)

[22. 延伸及扩展问题回答参考 38](#_Toc11916_WPSOffice_Level2)

[23. 项目中体现经验的点 38](#_Toc21778_WPSOffice_Level2)

[24. 论坛参考 38](#_Toc8464_WPSOffice_Level2)

**[五十八、 谈谈对vuexcookie对的看法？](#_Toc12470_WPSOffice_Level1)** **[40](#_Toc12470_WPSOffice_Level1)**

[25. 通常解法](#_Toc23320_WPSOffice_Level2) [40](#_Toc23320_WPSOffice_Level2)

[26. 通用大牛级解法](#_Toc31977_WPSOffice_Level2) [40](#_Toc31977_WPSOffice_Level2)

[27. 解法对比及优缺点](#_Toc27109_WPSOffice_Level2) [41](#_Toc27109_WPSOffice_Level2)

[28. 延伸及扩展问题回答参考](#_Toc9491_WPSOffice_Level2) [41](#_Toc9491_WPSOffice_Level2)

[29. 。项目中体现经验的点](#_Toc16599_WPSOffice_Level2) [41](#_Toc16599_WPSOffice_Level2)

[30. 论坛参考](#_Toc1897_WPSOffice_Level2) [42](#_Toc1897_WPSOffice_Level2)

**[五十九、 说一下promise？](#_Toc31030_WPSOffice_Level1)** **[42](#_Toc31030_WPSOffice_Level1)**

[25. 通常解法](#_Toc12079_WPSOffice_Level2) [42](#_Toc12079_WPSOffice_Level2)

[26. 通用大牛级解法](#_Toc6095_WPSOffice_Level2) [42](#_Toc6095_WPSOffice_Level2)

[27. 解法对比及优缺点](#_Toc3590_WPSOffice_Level2) [45](#_Toc3590_WPSOffice_Level2)

[28. 延伸及扩展问题回答参考](#_Toc27469_WPSOffice_Level2) [45](#_Toc27469_WPSOffice_Level2)

[29. 项目中体现经验的点](#_Toc25378_WPSOffice_Level2) [45](#_Toc25378_WPSOffice_Level2)

[30. 论坛参考](#_Toc7769_WPSOffice_Level2) [45](#_Toc7769_WPSOffice_Level2)

**[六十、 说一下jsonp？](#_Toc8464_WPSOffice_Level1)** **[45](#_Toc8464_WPSOffice_Level1)**

# SPA开发打包生成的bundle太大如何解决

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 65% | 3星 | 打包上线 | tree shaking | webpack |

## 通常解法

SPA开发，所有的文件都通过一个入口引用，通过webpack处理，最终将所有的依赖文件整合到一个文件里面，已达到上线的目的，但是将所有文件合在一起会导致整个包过大，会影响页面首次加载的时间。通常做法是使用webpack的UglifyJsPulign插件对生成的bundle.js进行压缩，去掉无效的代码和多余的注释，并通过extractTextPlugin将css文件从bundle中解析出来单独引用，来实现先bundle.js 文件的缩小。这个能在很大程度上减小bundle，是一个可行的办法。

## 通用大牛级解法

上面的解法可以作为基础，如果想进一步优化bundle.js，尤其是即使经过压缩和css抽取，文件依然很大的时候，光有上面的操作是不够的。这个时候还可以采取的比较高级的做法是使用ES6新的语法动态import，结合webpack的代码拆分，实现指定功能模块单独成包，也就是从bundle中再将一部分功能代码抽取出来，在需要用的时候让浏览器去加载，从而减小一开始页面引用文件的大小。

另外还可以考虑webpack的辅助插件AggressiveSplittingPlugin，CommonsChunkPlugin对代码进一步优化处理，以实现最终bundle代码的优化。

## 解法对比及优缺点

通常解法使用较多，配置也相对简单，只要对webpack有基本的使用经验都可以较快上手，如果出现问题一般百度就能解决。但是大牛解法使用较少，配置偏复杂，考虑问题比较多，不容易把控，需要精通webpack的使用才能得心应手

**综上所述**：SPA包的优化是一个渐进学习和积累的过程，需要平时多总结多练习

## 延伸及扩展问题回答参考

**问题**： 如果打包出现问题，一般怎么解决

**解答**： 1、针对问题查阅webpack文档

2、如果很难解决考虑优雅降级，先走通流程

## 项目中体现经验的点

对通常解决方案了熟于心，对大牛解法也能说出一二来

## 论坛参考

https://webpack.js.org

# 说说React生命周期？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 90% | 4星 | react | react核心概念 | 组件化 |

## 通常解法

React的生命周期相比于vue要更加有深度，名称叫起来也比较长，不是很容易记忆，尤其是react生命周期的使用，有很多细节可以体现出开发人员对react的理解程度。比如他的周期官方一般指：

创建期：

1. componentWillMount 组件DOM创建之前
2. componentDidMount 组件DOM创建完成

存在期：

1. componentWillReceiveProps 组件接收新props数据之前
2. shouldComponentUpdate 组件是否应该更新
3. componentWillUpdate 组件更新之前
4. componentDidUpdate 组件更新之后

销毁期：

1. componentWillUnmount 组件即将销毁

## 通用大牛级解法

在上面的基础加上：

constructor 初始化阶段

render 渲染阶段

另外解释具体的周期函数特点比如：

shouldComponentUpdate

这个函数可以接受三个参数分别是nextProps,nextState,nextContext，此函数可以省略不写，但是如果写就必须返回一个boolean，这点跟其他周期函数不同。我们可以利用这个函数对组件前后历史的数据进行对比从而实现组件性能优化，就是说通过对比发现数据没有真正更新就阻止周期函数继续运行，如果过发现数据有更新则允许周期继续运行。但需要提醒的是，该函数虽然可以用来做优化，如果不是确定能带来优化效果，尽量不要人为设置，react自身的性能就已经很高，应该优先考虑利用react自身的能力。

社区提供了第三方工具比如：

react-addons-pure-render-mixin

immutable.js

可以让我们结合shouldComponentUpdate对项目进一步进行优化，减少组件不必要的更新。另外react-addons-pure-render-mixin该工具是早期react优化的方案之一，但是现在新版的react已不再需要该工具，取而代之的是react提供的PureComponent组件，该组件跟普通组件相比没有shouldComponentUpdate函数，而是自动对前后历史数据进行浅对比，来阻止不必要的更新。但由于PureComponent进行的是浅对比，所以如果组件数据嵌套引用类型的话，会导致出现bug，这个时候就需要结合immutable来解决问题

## 解法对比及优缺点

通常解法，是对react特点的基本阐述，一般使用也已足够。但是如果因为业务太复杂，逻辑较深，频繁进行周期运行从而导致页面性能下降，这个时候就该考虑结合shouldComponentUpdate函数进行优化，只是这样会增加业务复杂度，而且使用不当反而会适得其反。所以也需要慎用。

## 延伸及扩展问题回答参考

**问题**： 在react中如何获取DOM，ref的使用有什么限制？

**解答**： 1、react是mvvm框架，优先考虑数据驱动视图，如果真的需要操作dom，可以使用ref属性，也可以使用findDOMNode函数。需要注意的是，操作DOM必须在componentDidMount之后才可以，因为这个周期之前DOM尚未生成，操作会报错。

Ref属性的值可以是字符也可以是函数，官方推荐函数，相对来说，后一种方式操作DOM更加可靠。

findDOMNode函数则可以从组建对象中提取dom节点，在有些场景下也可以使用。

2、另外可以考虑使用createPortal函数来优化需要操作dom的场景，该函数一般用于制作react插件。

## 项目中体现经验的点

对react周期函数的理解深度，掌握每一个函数的特点和应用场景

## 论坛参考

https://reactjs.org

# 如何实现发布订阅模式

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 90% | 4星 | Javascript编程模式 | 发布订阅模式 | Javascript |

## 通常解法

利用发布订阅模式可以产生更灵活、更松散耦合的系统；在实现该模式的关键就是理解几个概念：发布动作、订阅动作、订阅者；

整个流程就是，订阅动作会使订阅者监听某个发布动作，当订阅动作执行完毕后，所有的订阅者都处于监听状态；当发布动作执行后，监听该发布动作的所有监听者都会收到信号并且自身做出反应；

在以上的例子中，我们可以利用javaScript中的函数去充当一个订阅者。用一个数组来作为存放所有监听者的容器。开发一个方法用来模拟订阅动作，即把订阅者一次放入容器中，然后在开发一个方法用来模拟发布动作。即执行容器中的所有 函数；实现如下：



上述代码，listen方法就是一个订阅动作，tigger方法就是有一个发布动作；

放在list中的所有函数就是订阅者；当订阅动作可多次执行，当订阅动作执行后，订阅者就处于监听中。

## 通用大牛级解法

在理解了基本的解法和思路之后，我们应该考虑更多的情况和实用性…、

我们改进有以下几点：

* + - 1. 要在开发中使用该模式，要做好封装，采用面向对象的方式
      2. 要能够监听不同的类型；
      3. 在发布动作触发的时候，要能够给所有的订阅者传递信息

采用面向对象的方式的话，我们需要确定一个实例中应该有哪些属性和方法，我们需要一个list属性用来存储所有的订阅者，但是这里的list不是一个数组，应该是有一个对象，因为我们需要对区分不同类型的订阅者，该对象中的属性名将作为订阅者的类型。还需要有listen方法，该方法用来执行订阅动作，这里的订阅动作不仅添加订阅者，还需要区分类型，因此会接收两个参数。一个字符串，一个函数。再需要一个发布动作trigger，该方法需要接收一个字符串和一个载荷作为参数，字符串决定了触发何种类型的订阅者，载荷可以作为传递给各个订阅者的参数；实现如下：



## 解法对比及优缺点

发布者不需要知道订阅者的数量，订阅者听得话题或者订阅者是通过什么方式运行的。他们能够相互独立地运行，这样就可以让你分开开发这两部分而不需要担心对状态或实现的任何细微的影响。、

* 可扩展性

发布/订阅模式可以让系统在无论什么时候无法负载的时候扩展

* 更干净地设计

充分地利用好发布/订阅模式，你不得不深入地思考不同的组件是如何交互的。这通常会让我们有更干净地设计因为我们对解耦和松耦合的强调。

* 灵活性

你不需要担心不同的组件是如何组合在一起的。只要他们共同遵守一份协议

* 容易测试

你可以很好地找出发布者或订阅者是否会得到错误的信息

## 延伸及扩展问题回答参考

发布订阅模式可以解耦代码，那么它 有哪些劣势呢？

* 中间人也许不会通知系统消息传送的状态。所以我们无法知道消息传送是成功的还是失败的。紧耦合是需要保证这一点的。
* 发布者不知道订阅者的状态，反之亦然，这样的话，你根本不知道在另一端是否会没有问题？
* 随着订阅者和发布者数量的增加，不断增加的消息传送回导致架构的不稳定，容易在负载大的时候出问题
* 攻击者（恶意的发布者）能够入侵系统并且撕开它。这会导致恶意的消息被发布，订阅者能够获得他们以前并不能获得的消息。
* 更新发布者和订阅者的关系会是一个很难的问题，因为毕竟他们根本不认识对方。
* 需要中间人/代理商，消息规范和相关的规则会给系统增加一些复杂

## 项目中体现经验的点

关联行为场景，需要注意的是，关联行为是可拆分的，而不是“组合”关系。

事件多级触发场景。

跨系统的消息交换场景，如消息队列、事件总线的处理机制

# 封装一个全局的模态框

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 70% | 4星 | 面向对象 | javaScript | 单例模式 |

## 通常解法

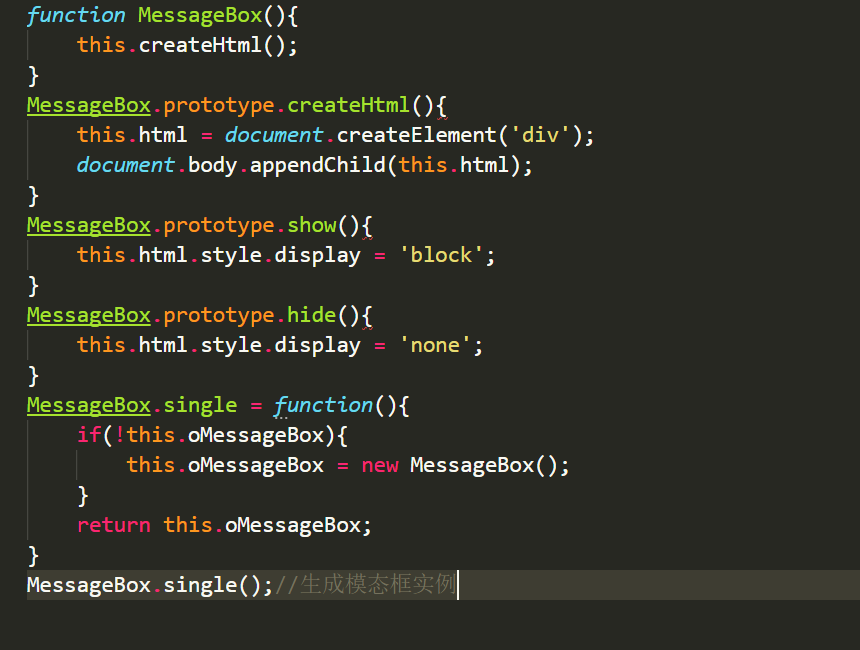
使用面向对象的方式，把模态框的行为模拟成一个类。在需要的时候实例好化该类，以达到该效果。具体在该类中，应该有生成模态框的html的方法。还需要有控制该模态框显示和隐藏的方式；这样的话全局就仅有一个模态框，以达到封装的目的；

实现代码如下：



## 通用大牛级解法

以上述思路为基础，我们发现有个问题，每次实例化的时候都会生成新的实例，同时也会给页面添加模态框，造成性能浪费；因此我们可以使用单例模式来解决上述问题，及该类智能实例化出一个对象；实现代码如下：



## 解法对比及优缺点

单例模式让我们在也面中的模态框有且仅有一个，而且如果在页面中没有弹出模态框的操作的话，根本不会实例化。在有弹出模态框操作的时候才会实例化，且仅仅是一次。在一定程度上优化来了性能；

## 延伸及扩展问题回答参考

什么是单例模式，是什么惰性单例？

* 单例模式的定义：保证一个类仅有一个实例，并提供一个访问他的全局访问点
* 单例模式的核心：是确保只有一个实例，并提供全局访问
* 惰性单例：在需要的时候才创建的对象实例
* 用途：在页面中有两个按钮，点击的时候需要显示响应弹窗并加载相应的css文件

## 5.项目中体现经验的点

在js中，我们经常会把全局变量当做单例模式来使用，例如：

var a={};

为什么a可以当做全局变量来使用呢，因为其满足以下两个条件：

1、对象a独一无二

2、a定义在全局作用域下，提供了全局访问

注：但是在js中建议使用命名空间，来减少全局变量的数量

## 6.论坛参考

<https://blog.csdn.net/wangxiuyan0228/article/details/79457538>

<https://www.cnblogs.com/haorui/p/3583337.html>

<https://blog.csdn.net/victor_e_n_01185/article/details/72875669>

# 如何解决vue组件之间的通讯?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 80% | 4星 | Vue框架 | 组件通讯 | 数据管理 |

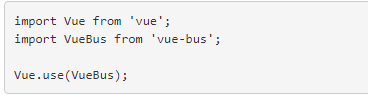
## 通常解法

在组件当中借助第三方插件 vue-bus

首先基于node下载vue-bus的依赖包：

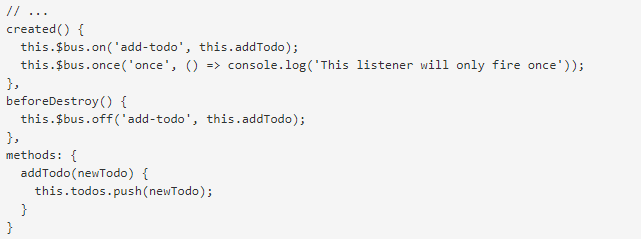
image.png

结合一个模块系统使用时，你需要通过Vue.use()来安装这个总线:

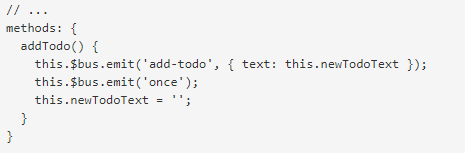


具体使用方法如下：

监听并清除



触发



## 通用大牛级解法

使用vuex实现组件之间数据的共享。

首先基于node下载vuex 的依赖包：

npm install vuex --save-dev

你需要通过Vue.use()来安装这个总线，然后实例化Vuex.Store()

import Vue from "vue"

import Vuex from "vuex"

Vue.use( Vuex )

export default new Vuex.Store({})

具体使用方法如下：

定义：

state：存放状态 getters：计算属性 mutations：更改state的方法 actions：提交mutations

import Vue from "vue"

import Vuex from "vuex"

Vue.use( Vuex )

export default new Vuex.Store({

state: { userInfor : '' },

getters: {

GetUserInfor: ( state ) => {

return state.userInfor

},

}

mutations: {

SET\_USET\_INFOR: (state, userInfor) => {

state.userInfor = userInfor

}

},

actions: {

SetUserInfor: ( { state,commit }, config ) => {

commit(‘SET\_USET\_INFOR’,config)

}

}

})

使用

computed:{

userInfor: ()=>{

return this.$store.getters.GetUserInfor //获取数据

}

},

methods: {

setUserInfor () {

this.$store.dispatch('SetUserInfor',{

name:’小明’,

age:12

}) ／／修改数据

},

}

## 解法对比及优缺点

vue-bus与vuex相比，各有各的优势，针对项目的不同可以选择对应的方案

1、 vue-bus的通信机制没有模块概念，如果管理内容过多容易混乱，而vuex针对于不同的业务需求可以对数据进行模块管理 。

2、 vue-bus主要是事件通信机制，而vuex则是状态管理针对于项目中的整个数据进行统一管理，划分比较明确。

3、 vue-bus理解使用起来比较简单，但是后期维护起来比较麻烦，容易造成命名冲突，而vuex则理解起来比较复杂，但是后期项目维护比较清晰。

4、 vue-bus适合坐简单项目中的组件通信机制，而vuex则比较适合做大型管理类项目

## 延伸及扩展问题回答参考

## 项目中体现经验的点

数据操作较多的大型项目

## 论坛参考

https://vuex.vuejs.org/zh/guide/

# 在主流的vue或react项目中跟后端的数据交互问题?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 80% | 4星 | 项目架构 | 后端交互 | 数据请求 |

## 通常解法

目前主流的数据请求方式有两种

* + - 1. axios第三方包，npm下载之后 直接使用
      2. fetch请求方式，考虑浏览器兼容问题 也需要下载第三方包。

使用方式 将下载好的包直接在组件逻辑内调用数据请求方法，然后获取数据之后进行数据操作

## 通用大牛级解法

考虑到大型项目中会有多处跟后端的数据交互所以总结出几点

* + - 1. 重复调用一个接口时会造成很多冗余代码
      2. 没有合理的逻辑api封装会造成代码过乱不容易维护
      3. 没有完整的数据请求方法调用起来会比较复杂

应该根据项目业务需求封装符合项目逻辑的一些公用请求方法如 request.js

将请求接口的方法封装为api.js文件

在项目的组件层只能通过调用action动作或者 api文件内的请求方法来实现跟后端的数据交互

## 解法对比及优缺点

本题考核的主要是对项目的数据交互方面的问题。

通用的解法只能满足项目的需求，不能实现对项目后期维护的考虑

大牛解法考虑后期项目维护及新开发人员加入时的便利性

## 延伸及扩展问题回答参考

**对后端请求逻辑的理解，比如请求头信息，及后端返回数据后对数据的批量处理**

## 项目中体现经验的点

请求头，数据过滤，公用逻辑封装。

## 论坛参考

https://www.jianshu.com/p/df464b26ae58

# 你是如何解决前端开发数据作用域问题?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 95% | 3星 | 前端 | 数据通讯 | 数据通讯 |

## 通常解法

通过前端浏览器存储；

比如cookie，sessionstorage，localstortage,等前端存储方式；

Sessionstorage.setItem(“userid”,”12345”)

## 通用大牛级解法

把上面的解法说一遍………

使用全局数据中心，比如flux，redux等前端全局数据中心思想；

实现原理，观察者模式加全局数据中心；

import Store from "../tool/flux"

// 全局数据中心

let initState={

num:1

}

export let actions={

addNum(text){

return {

type:"addNum",

text:text

}

}

}

let reducer = (state = initState, action) => {

switch (action.type) {

case 'addNum':{

let num=state.num+1

return {...state,...{num}}

}

default:

return state

}

}

export default new Store({

state:initState,

reducer

})

## 延伸及扩展问题回答参

前端存储的优缺点

前端存储的优点是：操作简单，api易学，进行数据的操作随意性比较大；方便在前端任何领域进行通讯；

前端存储的缺点则是：

对于数据的存储量有限，不利于大量数据的存储，对于数据的读写能力一般，不如直接操作js对象

前端全局数据中心与前端存储相比，无疑更为先进、方便和可靠。更利于数据的操作等等

## 项目中体现经验的点

**问题**：flux架构思想，前端数据流，前端mvc思想

**解答**： 1、基于前端数据交互混乱，以及数据追踪困难等问题提出的思想

2、基于组件化开发，导致出现了数据通讯问题

3、前端mvc思想，是仿照传统的服务端mvc进行改进而成，实现视图，控制层和数据逻辑层的分离，方便数据的追踪

项目中体现经验的点

做过组件化开发，主要体现接触过客户端服务端分离开发的经验，适合大多中级工程师必须掌握的知识点

## 论坛参考

<http://www.redux.org.cn/>

<https://cnodejs.org/topic/56599f64b31692e827fdcf96>

# 什么是同源策略

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 70% | 4星 | 前端 | 数据安全 | 数据安全 |

## 通常解法

同源策略（Same origin policy）是一种约定，它是[浏览器](https://baike.baidu.com/item/%E6%B5%8F%E8%A7%88%E5%99%A8/213911" \t "_blank)最核心也最基本的安全功能，如果缺少了同源策略，则浏览器的正常功能可能都会受到影响。可以说Web是构建在同源策略基础之上的，浏览器只是针对同源策略的一种实现。

同源策略，它是由[Netscape](https://baike.baidu.com/item/Netscape/2778944" \t "_blank)提出的一个著名的[安全策略](https://baike.baidu.com/item/%E5%AE%89%E5%85%A8%E7%AD%96%E7%95%A5/2890163" \t "_blank)。

现在所有支持JavaScript 的浏览器都会使用这个策略。

所谓同源是指，域名，协议，端口相同。

当一个浏览器的两个tab页中分别打开来 百度和谷歌的页面

当浏览器的百度tab页执行一个脚本的时候会检查这个脚本是属于哪个页面的，

即检查是否同源，只有和百度同源的脚本才会被执行。[1]

如果非同源，那么在请求数据时，浏览器会在控制台中报一个异常，提示拒绝访问。

## 通用大牛级解法

加上通用解法后

 它的精髓很简单：它认为自任何站点装载的信赖内容是不安全的。当被浏览器半信半疑的脚本运行在沙箱时，它们应该只被允许访问来自同一站点的资源，而不是那些来自其它站点可能怀有恶意的资源。

**为什么要有同源限制？**

      我们举例说明：比如一个黑客程序，他利用IFrame把真正的银行登录页面嵌到他的页面上，当你使用真实的用户名，密码登录时，他的页面就可以通过Javascript读取到你的表单中input中的内容，这样用户名，密码就轻松到手了。

**Ajax应用：**

      在Ajax应用中这种安全限制被突破。

      在普通的Javascript应用中，我们可以修改Frame的href，或者IFrame的src，以实现GET方式的跨域提交，但是却不能访问跨域的Frame/IFrame中的内容。

      而Ajax它通过XMLHTTP进行异步交互，这个对象同样能够与远程的服务器进行信息交互，而且更加危险的是，XMLHTTP是一个纯粹的Javascript对象，这样的交互过程，是在后台进行的，不被用户察觉。因此，XMLHTTP实际上已经突破了原有的Javascript的安全限制。

      如果我们又想利用XMLHTTP的无刷新异步交互能力，又不愿意公然突破Javascript的安全策略，可以选择的方案就是给XMLHTTP加上严格的同源限制。这样的安全策略，很类似于Applet的安全策略。IFrame的限制还仅仅是不能访问跨域HTMLDOM中的数据，而XMLHTTP则根本上限制了跨域请求的提交。

**浏览器支持：**

      而IE其实给这个安全策略开了两个想当然的后门，一个是：他假设你的本地文件，自然清楚将会访问什么内容，所以任何你的本地文件访问外部数据， 都不会收到任何的警告。另一个是：当你访问的网站脚本打算访问跨域的信息时， 他居然仅仅是弹出一个对话框来提醒你一下。如果一个欺诈网站，采用这样的手   
段，提供一个假页面给你，然后通过XMLHTTP帮你远程登录真实的银行服务器。只要10个用户里，有一个用户糊涂一下，点了一个确定。他们的盗取帐号行为，就成功了！ 你想想看，这是何等危险的事情！   
FireFox就不是这样的做法，缺省的情况下，FireFox根本就不支持跨域的XMLHTTP请求，根本就不给黑客这样的机会。

**避免同源策略：**

JSON和动态脚本标记

*<script type="text/javascript"  
            src="http://travel.com/findItinerary?username=sachiko&  
            reservationNum=1234&output=json&callback=showItinerary" />*

      当 JavaScript 代码动态地插入 <script> 标记时，浏览器会访问 src 属性中的 URL。这样会导致将查询字符串中的信息发送给服务器。在 清单 1中，所传递的是 username 和 reservation 作为名称值对传递。此外，查询字符串还包含向服务器请求的输出格式和回调函数的名称（即 showItinerary）。<script> 标记加载后，会执行回调函数，并通过回调函数的参数把从服务返回的信息传递给该回调函数。

Ajax代理

      Ajax 代理是一种应用级代理服务器，用于调解 Web 浏览器和服务器之间的 HTTP 请求和响应。Ajax 代理允许 Web 浏览器绕过同源策略，这样便可以使用 XMLHttpRequest 访问第三方服务器。要实现这种绕过，有如下两种方法可供选择：

* 客户端 Web 应用程序知道第三方 URL 并将该 URL 作为 HTTP 请求中的一个请求参数传递给 Ajax 代理。然后，代理将请求转发给 www.remoteservice.com。注意，可以把代理服务器的使用隐藏在 Web 应用程序开发人员所使用的 Ajax 库的实现中。对于 Web 应用程序开发人员而言，它看上去可能完全不具有同源策略。

客户端 Web 应用程序不知道第三方 URL，并且尝试通过 HTTP 访问 Ajax 代理服务器上的资源。通过一个预定义的编码规则，Ajax 代理将 所请求的 URL 转换为第三方服务器的 URL 并代表客户检索内容。这样一来，Web 应用程序开发人员看上去就像是在和代理服务器直接进行通信。

## 解法对比及优缺点

通常解法，仅就问题本身做简要解答，未能够对问题做详尽分析及解决。通用级解法，将同源策略的应用范围和应用场景，都有详细的介绍，进一步说明对此问题的深度理解。

## 延伸及扩展问题回答参考

**问题**： 接触到的前端同源策略的问题都有哪些？

解答：

（1） Cookie、LocalStorage 和 IndexDB 无法读取。

（2） DOM 无法获得。

（3） AJAX 请求不能发送。

问题：同源策略问题？

解答：

1. iframe
2. window.name
3. window.postMessage

## 项目中体现经验的点

当在项目中，解决数据的存储以及获取，还有通信时，经常遇到的问题；使用频率比较高

## 论坛参考

<http://www.360doc.com/content/17/0925/20/47820059_690133713.shtml>

<http://www.ruanyifeng.com/blog/2016/04/same-origin-policy.html>

<https://blog.csdn.net/tim_tsang/article/details/46124527>

# Generator async promise 的区别?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 95% | 3星 | Es6 | 异步 | Es6异步操作 |

## 通常解法

11Generator async promise 这三个api都是es6新增api，

Generator 函数定义需要通过\*来定义，通过yield表达式定义不同的状态，调用的时候需要使用next()进行状态的变更，直到遇到return后才能终止

Async是Generator函数的语法糖，也就是简写，但是Async中定义状态使用await定义状态，执行的时候不需要next，自动执行，并且async是结合promise一起使用的，await 的结果可以自动获取promise的resolve值

Promise是为了解决之前利用回调函数进行异步调用的解决方法，promise可以使用链式调用的方式得到成功的回调，并且可以在返回一个promise继续调用，

在promise中有3个状态进行中，成功，失败，可以使用resolve，reject，手动控制成功失败的回调

## 通用大牛级解法

1Generator async promise 这三个api都是es6新增api，也都可以理解为异步的解决方案，但是他们的适用场景不一样

先说Generator：

简单点可以把它理解成，Generator 函数是一个状态机，封装了多个内部状态。

执行 Generator 函数会返回一个遍历器对象，也就是说，Generator 函数除了状态机，还是一个遍历器对象生成函数。返回的遍历器对象，可以依次遍历 Generator 函数内部的每一个状态。

形式上，Generator 函数是一个普通函数，但是有两个特征。一是，function关键字与函数名之间有一个星号；二是，函数体内部使用yield表达式，定义不同的内部状态

而Generator在适用场景上我们可以把它理解为任务的挂起，在js中我们使用没有任务挂起这个概念的，直到Generator的出现，在Generator的使用上我们通过yield定义不同的状态，执行的时候需要使用next进行不同状态的调用，但是他是从上到下的，也就是只要需要yield就会停止执行后面的语句，调用next后继续执行后面的语句，在实际项目开发中，我经常会遇到这样的需求，填写用户信息的时候一般都是分成几步，这时候我们就可以利用Generator的任务挂起特性进行，不同步骤的数据存储。

然后就是promise

promise解决了传统异步操作回调函数更加合理强大，有了promise就可以将异步操作以同步操作的流程表达出来，避免了层层嵌套的回调函数。此外，Promise对象提供统一的接口，使得控制异步操作更加容易。

例如我常见的一个需求

有两个接口A,B, 他们之间有依赖关系，B接口必须等到A接口的值返回成功后，拿到A接口里面的一个属性才能请求，如果利用之前的回调函数形势的话就得是层层嵌套，代码维护比较麻烦，如果利用promise的话我们可以用链式调用的形势直观的表示出这种关系

而Async是Generator函数的语法糖，Async的出现主要是为了解决另外一个问题，刚刚说了Generator是解决任务的挂起，promise是解决异步回调问题，而async就是吧异步的操作，变成队列模式，

因为async中使用await做状态的定义，调用的时候不需要next（），自动执行，并且会讲每个await中promise中resolve结果赋值给await的变量上以供后面的步骤使用，每一个await都会等到promise返回结果后才会继续自动往下执行，这样就实现了我在日常生活中排队执行的概念，将所有的异步任务以同步的方式定义，不需要担心那个快那个慢，因为她是一个一个自动向下的

## 解法对比及优缺点

**综上所述**：第一种只是说出了 Generator async promise如何使用，而第二种说出了执行原理，存在的意义，适用的场景

## 延伸及扩展问题回答参考

**问题**： 如何合并多个请求，

**解答**：使用promise.all可以将多个使用promise定义的ajax请求合并处理

## 项目中体现经验的点

对于 Generator async promise正确的使用，利用它们不同的优点，实现不同的业务需求

## 论坛参考

[http://es6.ruanyifeng.com/#docs/promise](http://es6.ruanyifeng.com/" \l "docs/promise)

[http://es6.ruanyifeng.com/#docs/generator](http://es6.ruanyifeng.com/" \l "docs/generator)

[http://es6.ruanyifeng.com/#docs/async](http://es6.ruanyifeng.com/" \l "docs/async)

# 说一下前端存储?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 90% | 4星 | Javascript | 数据存储 | 存储 |

## 通常解法

Cookie技术浏览器兼容性好，但操作比较复杂，需要程序员自己封装，源生的Cookie接口不友好, 存储的内容较小， cookie的数据会随着ajax的请求发送到服务端，一般情况主要用在用户登录的时候我们可以通过在 Cookie 中存入一段辨别用户身份的数据，用于后台判断。

WebStorage则不能超过8MB，操作简单；可以代替一些cookie的工作，一般主要是用于存储一些本地数据，购物车数据之类的在安全方面的话，都不安全，一般就是对数据进行一些简单的加密，如base64编码，加密约定之类的东西localstorage、sessionstorage一个是长期存储，一个是会话存储

。

## 通用大牛级解法

**常用浏览器存储方案有，cookie，session，localstorage，sessionstroage**

**cookie**

cookie 本身不是用来做服务器端存储的，它是设计用来在服务器和客户端进行信息传递的，因此我们的每个 HTTP 请求都带着 cookie。但是 cookie 也具备浏览器端存储的能力（例如记住用户名和密码，也就是常用的登录功能），因此就被开发者用上了。

使用起来也非常简单，document.cookie = ....即可。

但是 cookie 有它致命的缺点：

存储量太小，只有 4KB

所有 HTTP 请求都带着，会影响获取资源的效率

API 简单，需要封装才能用

所有的api请求都会携带cookie，所以cookie不太安全，使用的时候一般都需要做加密处理

session

session是服务端的会话存储技术，他的生存周期只是保持在浏览器打开，浏览器关闭这个阶段之中

在Session被创建之后，就可以调用Session相关的方法往Session中增加内容了，而这些内容只会保存在服务器中，发到客户端的只有Session id；当客户端再次发送请求的时候，会将这个Session id带上，服务器接受到请求之后就会依据Session id找到相应的Session，从而再次使用之。正式这样一个过程，用户的状态也就得以保持了。

在使用session的时候我们一般都保存一些临时数据，常见的需求一般就是获取验证码

**locationStorage 和 sessionStorage**

后来，HTML5 标准就带来了sessionStorage和localStorage，先拿localStorage来说，它是专门为了浏览器端缓存而设计的。其优点有：

存储量增大到 5MB

不会带到 HTTP 请求中

API 适用于数据存储 localStorage.setItem(key, value) localStorage.getItem(key)

sessionStorage的区别就在于它是根据 session 过去时间而实现，而localStorage会永久有效，应用场景不同。例如，一些需要及时失效的重要信息放在sessionStorage中，一些不重要但是不经常设置的信息，放在localStorage中。

但是sessionstroage有个致命的缺点，就是无法多标签页共享。

## 解法对比及优缺点

通常解法，解释了cookie，本地存储的用法。通用级解法，不仅仅讲解了用法，还说出了其运行原理，以及不同技术之间的使用场景

## 延伸及扩展问题回答参考

**问题**： sessionstroage如何实现多标签页面数据共享

## 项目中体现经验的点

登录判断，本地数据线上数据的同步。

## 论坛参考

https://www.cnblogs.com/lonelydreamer/p/6169469.html

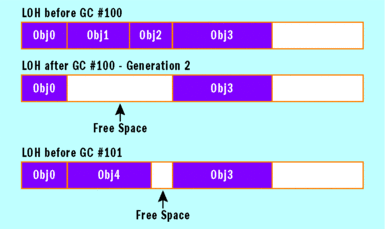
<https://www.cnblogs.com/andy-zhou/p/5360107.html>

# 解释一下javaScript中的垃圾回收机制？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 95% | 3星 | Js底层原理 | javaScript | 发布订阅 |

## 通常解法

javaScript具有自动垃圾收集机制，也就是说，执行环境会负责管理代码执行过程中使用的内存。而在C和C++之类的语言中，开发人员的一项基本任务就是手工跟踪内存的使用情况，这是造成许多问题的一个根源。在编写javaScript程序时，开发人员不用关心内存使用问题，所需内存的分配以及无用内存的回收完全实现了自动管理。这种垃圾收集机制的原理其实很简单：找出那些不再继续使用的变量，然后释放其占用的内存。为此，垃圾收集器会按照固定的时间间隔（或代码执行中预定的收集时间），周期性的执行这一操作；



如上图：javaScript操作的数据在内存中是这样存储的，当某块内存中存储的数据无用时，那么就要遵守javaScript的内存回收机制将这些空闲内存回收，以供重新分配；

## 通用大牛级解法

在通用解法的基础上，我们分一下函数中局部变量的生命周期。局部变量只在函数执行的过程中存在。而在这个过程中，会为局部变量在栈（或堆）内存上分配相应的空间，以便存储他们的值。然后在函数中使用这些变量，直至函数执行结束。此时，局部变量就没有存在的必要了，因此可以释放它们的内存以供将来使用。在这种情况下，很容易判断变量是否还有存在的必要；但并非所有情况下都这么容易就能得出结论。垃圾收集器必须跟踪哪个变量有用哪个变量没有，对于不再有用的变量打上标记，以备将来收回其占用的内存。用于表示无用变量的策略可能会因实现而异，但具体到浏览器中的实现通常有两个策略。分别是标记清除和引用计数

标记清除

javaScript中最常用的垃圾收集方式是标记清除（mark-and-sweep）。当变量进入环境时，就将这个变量标记为“进入环境”。从逻辑上讲，永远不能释放进入环境的变量所占用的内存，因为只要执行流进入相应的环境，就可能会用到它们。而当变量离开环境时，则将其标记为“离开环境”。

垃圾收集器在运行的时候会给存储在内存中的所有变量都加上标记。然后，它会去掉环境中的变量以及被环境中的变量引用的变量标记。而在此之后再被加上标记的变量将被视为准备删除的变量，原因是环境中的变量已经无法访问到这些变量了。最后，垃圾收集器完成内存清除工作，销毁那些带标记的值并回收他们所占用的内存空间。

引用计数

另一种不太常见的垃圾收集策略叫做引用计数。引用计数的含义是跟踪记录每个值被引用的次数。当声明了一个变量并将一个引用类型赋值给该变量时，则这个值的引用此时就是1。如果同一个值又被赋给另一个变量，则该值的引用次数加1。相反，如果包含对这个值引用的变量又取得了另外一个值，则这个值的引用次数减1。当这个值的引用次数变成0是，则说明没有办法再访问这个值了，因而就可以将其占用的内存空间回收回来。这样，当垃圾收集器下次再运行时，它就会释放那些引用次数为零的值所占用的内存。

## 解法对比及优缺点

对比以上解法，可以掌握以下几点：

* 基本类型值在内存中占据固定大小的空间，因此被保存在栈内存中；
* 从一个变量向另一个变量复制基本数据类型的值，会创建这个值的一个副本；
* 引用类型的值是对象，保存在堆内存中；
* 包含引用类型值的变量实际上包含的并不是对象本身，而是一个指向该对象的指针；
* 当代码中存在引用计数的策略时，循环引用会导致问题；

## 延伸及扩展问题回答参考

根据垃圾回收机制来思考，如何管理内存；

使用具备垃圾收集机制的语言编写程序，开发人员一般不必操心内存管理问题。但是，javaScript在进行内存管理及垃圾回收时面临的问题还是有点与众不同。其中最主要的一个问题，就是分配给web浏览器的可用内存数量通常比分配给桌面应用程序的少。这样做的目的主要是出于安全方面的考虑，目的是防止运行javaScript的网页耗尽全部系统内存而导致系统崩溃。内存限制问题不仅会影响给变量分配内存，同时还会影响调用栈以及在一个线程中能够同时执行的语句数量。

因此，确保占用最少的内存可以让页面获得更好的性能。而优化内存占用的最佳方式，就是为执行中的代码只保存必要的数据。一旦数据不再有用，最好通过将其值设置为null来释放其引用---这个叫做接触引用。这一做法适用于大多数全局变量和全局对象的属性。局部变量会在它们离开执行环境时自动被解除引用。

## 项目中体现经验的点

深刻理解垃圾回收机制，主要代码的规范和书写方式，尽量避免内存泄露的问题；

## 论坛参考

<https://www.jb51.net/article/75292.htm>

<https://blog.csdn.net/Christine95/article/details/50877645>

<https://blog.csdn.net/IamChuancey/article/details/78452366>

# 解释一下React中的JSX语法？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 75% | 3星 | 虚拟DOM | 开发效率 | js |

## 通常解法

JavaScript eXtension，或更常见的JSX，是一个React扩展，允许我们编写看起来像 HTML的JavaScript 。

可以在 JSX 中使用 JavaScript 表达式。表达式写在花括号 {} 中；

在 JSX 中不能使用 if else 语句，但可以使用 conditional (三元运算) 表达式来替代；

React 推荐使用内联样式。我们可以使用 camelCase 语法来设置内联样式. React 会在指定元素数字后自动添加 px；

JSX 允许在模板中插入数组，数组会自动展开所有成员；

JSX事件绑定使用驼峰写法，比如<div onClick={()=>{}}></div>

## 通用大牛级解法

虽然JSX看起来像HTML，但它实际上只是React.createElement()的语法糖。当组件渲染时，它输出一个React元素树或该组件输出的HTML元素的虚拟表示。React然后将基于此React元素表示来确定对实际DOM所做的更改。

例如：

class Header extends React.Component {

render() {

return (

<h1>Hello World</h1>

);

}

}

其实对应于

class HelloWorld extends React.Component {

render() {

return (

React.createElement(

'h1',

{className: 'large'},

'Hello World'

)

);

}

}

就是说JSX最终会被babel等编译工具转换为普通的JS来让浏览器识别，而转换的结果是React.createElement()函数，该函数不会生成真实的DOM而是创建一个虚拟DOM，虚拟DOM本身就是一个对象。ReactDOM.render()将虚拟DOM渲染到视图上，会经过diff运算以减小不必要的DOM操作，从而提高页面性能。

## 解法对比及优缺点

* 普通解法只是概括jsx表面特点，以及如何使用。
* 大牛级解法对jsx的实质作了进一步阐述，充分体现开发人员对jsx语法的理解

## 延伸及扩展问题回答参考

虚拟DOM的概念被react框架发明，得到了社区的积极响应，目前主流应用框架都引入了虚拟DOM的概念，比如VUE也通过虚拟DOM+diff运算来提升页面性能

## 项目中体现经验的点

深刻理解JSX语法的特点和原理；

## 论坛参考

http://www.runoob.com/react/react-jsx.html

https://blog.csdn.net/tianzhw/article/details/78812475

<https://www.cnblogs.com/zourong/p/6043914.html>

# redux三大原则？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 85% | 4星 | 数据管理 | 跨组建通信 | [reducers](http://www.redux.org.cn/docs/Glossary.html" \l "reducer) |

Redux 可以用这三个基本原则来描述：

## 单一数据源

**整个应用的 [state](http://www.redux.org.cn/docs/Glossary.html" \l "state) 被储存在一棵 object tree 中，并且这个 object tree 只存在于唯一一个 [store](http://www.redux.org.cn/docs/Glossary.html" \l "store) 中。**

这让同构应用开发变得非常容易。来自服务端的 state 可以在无需编写更多代码的情况下被序列化并注入到客户端中。由于是单一的 state tree ，调试也变得非常容易。在开发中，你可以把应用的 state 保存在本地，从而加快开发速度。此外，受益于单一的 state tree ，以前难以实现的如“撤销/重做”这类功能也变得轻而易举。

console.log(store.getState())

/\* Prints

{

visibilityFilter: 'SHOW\_ALL',

todos: [

{

text: 'Consider using Redux',

completed: true,

},

{

text: 'Keep all state in a single tree',

completed: false

}

]

}

\*/

## State 是只读的

**惟一改变 state 的方法就是触发 [action](http://www.redux.org.cn/docs/Glossary.html" \l "action)，action 是一个用于描述已发生事件的普通对象。**

这样确保了视图和网络请求都不能直接修改 state，相反它们只能表达想要修改的意图。因为所有的修改都被集中化处理，且严格按照一个接一个的顺序执行，因此不用担心 race condition 的出现。 Action 就是普通对象而已，因此它们可以被日志打印、序列化、储存、后期调试或测试时回放出来。

store.dispatch({

type: 'COMPLETE\_TODO',

index: 1

});

store.dispatch({

type: 'SET\_VISIBILITY\_FILTER',

filter: 'SHOW\_COMPLETED'

});

## 使用纯函数来执行修改

**为了描述 action 如何改变 state tree ，你需要编写 [reducers](http://www.redux.org.cn/docs/Glossary.html" \l "reducer)。**

Reducer 只是一些纯函数，它接收先前的 state 和 action，并返回新的 state。刚开始你可以只有一个 reducer，随着应用变大，你可以把它拆成多个小的 reducers，分别独立地操作 state tree 的不同部分，因为 reducer 只是函数，你可以控制它们被调用的顺序，传入附加数据，甚至编写可复用的 reducer 来处理一些通用任务，如分页器。

function visibilityFilter(state = 'SHOW\_ALL', action) {

switch (action.type) {

case 'SET\_VISIBILITY\_FILTER':

return action.filter

default:

return state

}

}

function todos(state = [], action) {

switch (action.type) {

case 'ADD\_TODO':

return [

...state,

{

text: action.text,

completed: false

}

]

case 'COMPLETE\_TODO':

return state.map((todo, index) => {

if (index === action.index) {

return Object.assign({}, todo, {

completed: true

})

}

return todo

})

default:

return state

}

}

import { combineReducers, createStore } from 'redux'

let reducer = combineReducers({ visibilityFilter, todos })

let store = createStore(reducer)

## **延伸及扩展问题回答参考**

## **项目中体现经验的点**

## **论坛参考**

# redux-saga中间件？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 40% | 5星 | redux中间件 | 异步流程管理了 | 数据容器 |

## 通常解法

Saga是一个可以用来处理复杂的异步逻辑的模块，并且由redux的action触发，Saga副作用就是在action触发reducer之后执行的一些动作， 这些动作包括但不限于，连接网络，io读写，触发其他action。并且，因为Saga的副作用是通过redux的action触发的，每一个action，saga都会像reducer一样接收到。并且通过触发不同的action, 我们可以控制这些副作用的状态， 例如，启动，停止，取消。通常配置Saga中间件由2部分组成：worker saga和watcher saga。watcher saga用于订阅action，worker saga用于根据action执行相应的操作。

所有的saga都是基于ES6的generator，generator函数可以将异步逻辑转为同步逻辑来编写。

例如：

watcher saga：function\* watch(){

yield takeEvery(‘action type’, worker saga)；

}

worker saga: function\* worker(){

yield call()

yield put()

}；

## 通用大牛级解法

Saga可以实现复杂异步流程管理，适合在大型项目且业务场景复杂的情况下使用。在业务逻辑层，可以简化代码，使代码更加容易阅读。 在重用方面，解耦显示层和业务层之后， 代码的重用度也得到了提升。使用基础API如take, put, call, fork, all等可以实现多流程控制，包括顺序执行，并行执行和竞争执行等。相比于redux-thunk中间件只是实现异步流程控制反转dispatch(function)来说，避免了难以测试的问题，逻辑流程也更加清晰。

Saga的应用简单总结来说就是：

视图=>dispatch(action)=>saga中间件:订阅action(watcher)+ worker + dispatch(action)=>数据容器redux(reducer)。

虽然saga可以起很大的效果，但是也有弊端：

* redux-saga模型的理解和学习需要投入很多精力
* 因为需要用action触发，所以会产生很多对于reducer无用的action, 但是reducer一样会跑一轮，虽然目前没有观测到性能下降，但还是有计算开销
* 在action的定义上要谨慎，避免action在saga和reducer之间重复触发，造成死循环
* 增加了项目目录结构复杂度和维护成本

。

## 解法对比及优缺点

普通解法只是对redux-saga基本使用的介绍，仅限于知道怎么用，不能说明redux-saga的适用场景，以及更复杂场景的使用情况。

大牛解法则清楚地说明了redux-saga的应用流程，心里明确什么时候需要使用saga。并且对潜在问题做了说明，同时对比了其他类似的中间件，体现了丰富的应用经验

## 延伸及扩展问题回答参考

对比说明redux-saga和redux-thunk的特点？

越是用来解决具体问题的技术，使用起来越容易，越高效，学习成本越低；越是用来解决宽泛问题的技术，使用起来越难，学习成本越高。 thunk解决的是很具体的一个问题，就是在action到达reducer之前做一些其他的业务，比如fetch后端, 它在做这件事的上很高效。而Saga解决的问题要更宽泛一些，因为saga只是拦截了action，至于做什么，开发者需要自己来考虑，可以是fetch后端，也可以是更新redux store, 甚至可以执行action带进来的callback。 很显然对于一个业务层来说,saga会是一个更合适的选择，但同时也带来了学习成本的提高。

## 项目中体现经验的点

深刻理解项目异步流程管理问题，以及选择正确的解决方案；

## 论坛参考

<https://redux-saga-in-chinese.js.org/docs/api/>

<https://www.jianshu.com/p/89ed2a01a3db>











# react-redux中的容器组件(container)和展示组件(component)？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 70% | 3星 | 全局状态管理 | 状态管理 | redux |

react-redux的作用是连接（connect）store和容器组件的。store是redux提供的，容器组件是react提供的。

## 组织应用的组件

* 组织应用的组件
  + 容器组件
  + 展示组件

容器组件：位于应用最顶层的组件，用来与redux连接的。从redux中获取数据作为props。

展示组件：位于应用的中间或者子组件，是纯粹的组件，与redux没有关系。他们从自己的父组件获取数据作为props，他们的共同根组件是应用的唯一的容器组件。展示组件可以维持少量的自身状态信息。

## 连接Store与组件

react-redux仅仅提供两个关键模块：Provider和connect。

源码：

import Provider from './components/Provider'

import connect from './components/connect'

export { Provider, connect }

* Provider：是一个组件，接受一个store属性和一个子组件（也就是上面说到的：store是redux提供的，容器组件是是react提供的。）

## 通常解法

ReactDOM.render(

<Provider store={store}>

<Handler routerState={routerState} />

</Provider>,

document.getElementById('root')

);

* connect([mapStateToProps], [mapDispatchToProps], [mergeProps], [options])：connect返回一个函数，它接受一个React组件的构造函数作为连接对象，最终返回连接好的组件构造函数。

import \* as actionCreators from './actionCreators'

function mapStateToProps(state) {

return { todos: state.todos }

}

export default connect(mapStateToProps, actionCreators)(MyRootComponent)

## 延伸及扩展问题回答参考

## 项目中体现经验的点

## 论坛参考

# 简述前端性能优化的措施？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 70% | 4星 | 性能优化 | javaScript | 性能优化 |

## 通常解法

**从网络加载方面来考虑**

* 减少HTTP请求次数

建议尽可能的根据需要去合并静态资源图片、JavaScript代码和CSS文件，减少页面请求数，这样可以缩短页面首次访问的等待时间，另外也要尽量的避免重复资源，防止增加多余的请求

* 减少HTTP请求大小

除了减少请求资源数，也要减少每个http请求的大小。比如减少没必要的图片，JS，CSS以及HTML等，对文件进行压缩优化，开启GZIP压缩传输内容，缩短网络传输等待延迟

* 将CSS和JS放到外部文件中，避免使用style和script标签引入

在HTML文件中引入外部的资源可以有效利用浏览器的静态资源缓存。有时候在移动端对请求数比较在意的会为了减少请求把CSS和JS文件直接写到HTML里边，具体根据CSS和JS文件大小和业务场景来分析。如果CSS和JS文件内容较多，逻辑比较复杂，建议放到外部引入

* 避免页面中空的href和src

当link标签的href属性为空，或者script、img、iframe标签的src属性为空的时候，浏览器在渲染的过程中还是会把href和src的空内容进行加载，直到加载失败。这样就阻塞了页面中其他资源的下载进程，并且最后加载的内容是无效的，因此要尽量避免。

* 为HTML指定Cache-Control或者Expires

为HTML指定Cache-Control或者Expires可以将HTML内容缓存起来，避免频繁向服务器发送请求。在页面Cache-Control或Expires头部又消失，浏览器会直接从缓存读取内容，不向服务器发送请求

* 静态资源不同域名存放

浏览器在同一时刻向同一个域名请求文件的并行下载数是有限的，因此可以理由多个域名的主机来存放不同的静态资源，增大页面加载时资源的并行下载数。

* 使用get请求

POST请求会首先发送文件头，然后发送HTTP正文的数据。而使用GET只发送头部，所以在拉取数据时使用GET请求效率更高

* 消除阻塞页面的CSS和JS

对于页面中加载时间过长的CSS或JS文件，需要进行合理的拆分或者延后加载，保证关键的资源能快速加载完成

* 按需加载

## 通用大牛级解法

不仅考虑到网络请求方面，同时也考虑到页面渲染方面

* 把CSS资源引用放到HTML文件顶部

这样浏览器可以优先下载CSS并尽快完成页面渲染

* JavaScript文件引用放到HTML文件底部

可以防止JavaScript的加载和解析执行对页面渲染造成阻塞。由于JavaScript资源默认是解析阻塞的，除非被标记为异步或者通过其他的方式异步加载，否则会阻塞HTML DOM解析和CSS渲染过程

* 不要在HTML中直接缩放图片

在HTML中直接缩放图片会导致页面内容的重排重绘，此时可能会使页面中的其他操作产生卡顿，因此要尽量减少在页面中直接进行图片缩放

* 减少DOM元素数量和深度

HTML中标签元素约的，标签的层级越深，浏览器解析DOM并绘制到浏览器中说花的时间就越长。

* 尽量避免使用table、iframe等慢元素

<table>内容的渲染是讲table的DOM渲染树全部生成完并一次性绘制到页面上，所以在长表格渲染时很耗性能，应该尽量避免使用，可以考虑用ul代替。尽量使用异步的方式动态的加载iframe，因为iframe内资源的下载进程会阻塞父页面静态资源的下载以及HTML DOM的解析

* 避免运行耗时的JavaScript

长时间运行的JavaScript会阻塞浏览器构建DOM树、DOM渲染树、渲染页面。所以任何与页面初次渲染无关的逻辑功能都应该延迟加载执行，这和JavaScript资源的异步加载思路一致

## 解法对比及优缺点

前端优化的策略有很多，主要包括网络加载，页面渲染，CSS优化，JS执行优化，缓存，图片，协议几大类；

网络请求是影响前端性能的一个主要因素，当然页面渲染也是一个非常重要的因素；因此我们应该针对这个两个方面都进行优化；

## 延伸及扩展问题回答参考

* 请减少HTTP请求基本原理：

在浏览器(客户端)和服务器发生通信时，就已经消耗了大量的时间，尤其是在网络情况比较糟糕的时候，这个问题尤其的突出。

一个正常HTTP请求的流程简述：如在浏览器中输入"www.xxxxxx.com"并按下回车，浏览器再与这个URL指向的服务器建立连接，然后浏览器才能向服务器发送请求信息，服务器在接受到请求的信息后再返回相应的信息，浏览器接收到来自服务器的应答信息后，对这些数据解释执行。

而当我们请求的网页文件中有很多图片、CSS、JS甚至音乐等信息时，将会频繁的与服务器建立连接，与释放连接，这必定会造成资源的浪费，且每个HTTP请求都会对服务器和浏览器产生性能负担。

网速相同的条件下，下载一个100KB的图片比下载两个50KB的图片要快。所以，请减少HTTP请求。

解决办法：

合并图片（css sprites），合并CSS和JS文件；图片较多的页面也可以使用 lazyLoad 等技术进行优化。

* 请正确理解 Repaint 和 Reflow

基本原理：

Repaint(重绘)就是在一个元素的外观被改变，但没有改变布局(宽高)的情况下发生，如改变visibility、outline、背景色等等。

Reflow(重排)就是DOM的变化影响到了元素的几何属性（宽和高），浏览器会重新计算元素的几何属性，会使渲染树中受到影响的部分失效，浏览器会验证DOM树上的所有其它结点的visibility属性，这也是Reflow低效的原因。如：改变窗囗大小、改变文字大小、内容的改变、浏览器窗口变化，style属性的改变等等。如果Reflow的过于频繁，CPU使用率就会噌噌的往上涨，所以前端也就有必要知道 Repaint 和 Reflow的知识。

减少性能影响的办法：

上面提到通过设置style属性改变结点样式的话，每设置一次都会导致一次reflow，所以最好通过设置class的方式； 有动画效果的元素，它的position属性应当设为fixed或absolute，这样不会影响其它元素的布局；如果功能需求上不能设置position为fixed或absolute，那么就权衡速度的平滑性。

总之，因为 Reflow 有时确实不可避免，所以只能尽可能限制Reflow的影响范围。

## 项目中体现经验的点

在操作页面的dom时，方式很多，但是我们应该从影响前端性能的这个方面着手去解决这个问题，避免过多的重排，操作dom元素不要过深

## 6.论坛参考

https://www.cnblogs.com/liulilin/p/7245125.html

<https://www.jianshu.com/p/ead7dab72cd6?mType=Group>

<https://blog.csdn.net/w2326ice/article/details/64122372>

# 你是如何解决移动端适配问题？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 95% | 3星 | 前端 | 移动端适配 | 移动端适配 |

## 通常解法

通过百分比进行单位的换算；

通过媒介查询进行单位的设置；

使用媒介查询配合rem

@media screen and (min-width:360px) and (max-width:374px) and (orientation:portrait) {

html { font-size: 703%; }

}

@media screen and (min-width:375px) and (max-width:383px) and (orientation:portrait) {

html { font-size: 732.4%; }

}

@media screen and (min-width:384px) and (max-width:399px) and (orientation:portrait) {

html { font-size: 750%; }

}

@media screen and (min-width:400px) and (max-width:413px) and (orientation:portrait) {

html { font-size: 781.25%; }

}

@media screen and (min-width:414px) and (max-width:431px) and (orientation:portrait){

html { font-size: 808.6%; }

}

@media screen and (min-width:432px) and (max-width:479px) and (orientation:portrait){

html { font-size: 843.75%; }

}

## 通用大牛级解法

把上面的解法说一遍………

**网易手机端**

body的width都是7.5rem。很明显，目前网易的手机端设计稿是基于iPhone6，750（设计师给的设计稿是物理分辨率，会是我们写样式的逻辑分辨率的两倍，如果给的设计稿是640，那么是基于iPhone5，320），且基准值是100px（750/7.5=100）。这个基准值很关键，后面的css换算，都和这个基准值有关。动态font-size： 我们看到图1、图2、图3的font-size都有根据屏幕大小而动态改变，可以推算出公式：

屏幕宽度/设计稿rem宽度=页面动态font-size值（如：375/7.5=50）

获取到这个值，再赋给html元素的style：

document.documentElement.style.fontSize = document.documentElement.clientWidth / 7.5 + ‘px‘;

这样就设置好了每个页面的根fonts-size，因为rem单位是基于根font-size，因此只要确定一种设计稿对应手机的换算，其余屏幕尺寸均可自动适配。

上面我们得出设计稿换算rem的基准值是100，因此只需要把设计稿上的px值除以100即为我们要的rem值。

> Px/100=rem，所以100px=1rem,25px=0.25rem

**淘宝手机端**

引入： 直接引用阿里的CDN文件（或下载到本地引入）

<script src="http://g.tbcdn.cn/mtb/lib-flexible/0.3.4/??flexible\_css.js,flexible.js"></script>

设定： 页面不要设定 。Flexible会自动设定每个屏幕宽度的根font-size、动态viewport、针对Retina屏做的dpr。

换算： 假设拿到的设计稿和上述网易的一样都是750，Flexible会把设计稿分为10份，可以理解为页面width=10rem，即1rem=75px，所以根font-size（基准值）=75px。

## 解法对比及优缺点

**通用方案**

1、设置根font-size：625%（或其它自定的值，但换算规则1rem不能小于12px）

2、通过媒体查询分别设置每个屏幕的根font-size

3、css直接除以2再除以100即可换算为rem。

优：有一定适用性，换算也较为简单。

劣：有兼容性的坑，对不同手机适配不是非常精准；需要设置多个媒体查询来适应不同手机，单某款手机尺寸不在设置范围之内，会导致无法适配。

**网易方案**

1、拿到设计稿除以100，得到宽度rem值

2、通过给html的style设置font-size，把1里面得到的宽度rem值代入x  document.documentElement.style.fontSize = document.documentElement.clientWidth / x + ‘px‘;

3、设计稿px/100即可换算为rem

优：通过动态根font-size来做适配，基本无兼容性问题，适配较为精准，换算简便。

劣：无viewport缩放，且针对iPhone的Retina屏没有做适配，导致对一些手机的适配不是很到位

**手淘方案**

1、拿到设计稿除以10，得到font-size基准值

2、引入flexible

3、不要设置meta的viewport缩放值

4、设计稿px/ font-size基准值，即可换算为rem

优：通过动态根font-size、viewpor、dpr来做适配，无兼容性问题，适配精准。

劣：需要根据设计稿进行基准值换算，在不使用sublime text编辑器插件开发时，单位计算复杂。

## 延伸及扩展问题回答参考

**问题**：移动端项目对于背景图的加载

**解答**： 使用的网易解决方案可以实现雪碧图的使用；

项目中体现经验的点

使用网易的解决的方案可以使用的移动端适配更加精确

## 论坛参考

<https://www.cnblogs.com/liangxuru/p/6970629.html>

<https://www.cnblogs.com/cench/p/5314044.html>

# 谈谈对node对的看法？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 70% | 4星 | 前端 | 数据安全 | 数据安全 |

## 通常解法

Node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行环境。   
Node.js 使用了一个事件驱动、非阻塞式 I/O 的模型，使其轻量又高效。   
Node.js 的包管理器 npm，是全球最大的开源库生态系统。

Node解决了js跑在服务端的一个空缺

## 通用大牛级解法

加上通用解法后

可以解决高并发，它是单线程，当访问量很多时，将访问者分配到不同的内存中，不同的内存区做不同的事，以快速解决这个线程。就像医院的分科室看病人。效率快，但消耗内存大、异步和事件驱动。概扩起来就三点：单线程、异步I/O、事件驱动。nodejs离不开ChormeV8引擎，也就是V8引擎是来解释javascript。用nodejs来搭建高性能的Web服务器，因此node.js是基于服务器端的javascript

node主要应用场景是在大前端，阿里的思路是比较合适的，但是必须要注意，绝对不能让node做太多的业务逻辑，他只适合接受人家生成好的数据，然后或渲染后，或直接发送到客户端。如果让node做复杂的业务逻辑，那会得不偿失的

回调模式下的异步是有明显缺陷的，程序的执行顺序必须依靠回调来保证，没有层层回调，就没有可以保障的逻辑顺序，这也就注定了，node不能做复杂的业务逻辑。javascript语言本身也一直在和回调做斗争，promise，generator都可以将回调包装起来，在代码的某个部分形成形式同步，但是这种模式进化的还不完全，还不能做到与回调完全割裂，做到完全的形式同步。但是形式同步肯定是发展的方向，这种模式即可以获得异步的好处，又可以有效回避回调带来的编程困难，在业务逻辑上可以更简单的表达。

就现在的环境来说，大家的思路还没转过弯，对回调的批评认为都是不好的，这些人是不敢面对现实，javascript都在变，这些人的脑子却不肯变，还以为回调就代表异步。

Node.js的发展给javascript注入了新的生命力

至于javascript的垢病，个人感觉，不在它的callback而在它的随意性，随意到想怎么写都行，但正是这点给它带来了惊人的开发效率，做好代码规范和文档工作可以减少javascript的随意性带来的负面影响。

WEB端、移动端、桌面端、甚至嵌入式，javascript已经无处不在。接下来，ES6的实现会让众多习惯同步或者不喜欢回调的开发人员能够更快地上手javascript写出符合他们思维习惯的代码，这些开发人员会是更大的群体，那么也许javascript会横扫应用开发也不一定。

所以，javascript很有前途，那Node.js自然就有前途。

## 解法对比及优缺点

大牛级解法：

优点：可以比较深入的解释清楚node底层原理和带来的一些良性循环；

缺点：如果对基础知识不牢固，可能会带来一系列的问题；

通用解法：

优点：可以比较概括的回答面试的问题

缺点：不够相信，不够全面

## 延伸及扩展问题回答参考

**问题**：目前前端对node的应用场景有哪些？

解答： 前端自动化，客户端服务都在大规模的使用node进行问题的解决！

## 。项目中体现经验的点

**可以解决前端本地开发的服务问题，可以实现接口的代理，可以实现客户端服务等**

## 论坛参考

<https://blog.csdn.net/hsj1669666567/article/details/79568545>

https://cnodejs.org/topic/53c88a83c9507b4044b0a78a

# Vue的生命周期？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 95% | 3星 | Es6 | 异步 | Es6异步操作 |

## 通常解法

vue生命周期一共有11个，

**beforeCreate,created**

**beforeMount,mounted**

**beforeUpdate, updated**

**actived** , deactivated

beforeDestroy, destroyed

errorCaptured

## 通用大牛级解法

vue的生命周期主要分为几个简单，数据初始化，dom挂载，数据更新，组件卸载，在一个就是开启了组件缓存的时候，会有组件启用和组件停用阶段，每个阶段都去前后两个钩子除了缓存的那俩

数据初始化阶段

beforeCreate：在实例初始化之后，数据观测 (data observer) 和 event/watcher 事件配置之前被调用。

created：实例已经创建完成之后被调用。在这一步，实例已完成以下的配置：数据观测 (data observer)，属性和方法的运算，watch/event 事件回调。然而，挂载阶段还没开始，$el 属性目前不可见。

dom挂载阶段

beforeMount：在挂载开始之前被调用：相关的 render 函数首次被调用。

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。

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 替换掉 mounted

数据跟新阶段

beforeUpdate：数据更新时调用，发生在虚拟 DOM 重新渲染和打补丁之前。

updated：由于数据更改导致的虚拟 DOM 重新渲染和打补丁，在这之后会调用该钩子。

当这个钩子被调用时，组件 DOM 已经更新，所以你现在可以执行依赖于 DOM 的操作。然而在大多数情况下，你应该避免在此期间更改状态。如果要相应状态改变，通常最好使用计算属性或 watcher 取而代之。

updated 不会承诺所有的子组件也都一起被重绘。如果你希望等到整个视图都重绘完毕，可以用 vm.$nextTick 替换掉 updated：

缓存启用的时候会有下面两个钩子

activated：keep-alive 组件激活时调用。

deactivated：keep-alive 组件停用时调用。

组件卸载的时候：

beforeDestroy：实例销毁之前调用。在这一步，实例仍然完全可用。

destroyed：Vue 实例销毁后调用。调用后，Vue 实例指示的所有东西都会解绑定，所有的事件监听器会被移除，所有的子实例也会被销毁。

## 解法对比及优缺点

**综上所述**：第一种只是说出了生命周期方法，而第二种说出生命周期的执行作用

## 延伸及扩展问题回答参考

**问题**： 数据请求应该在那个生命周期，

**解答**：在created里面

## 项目中体现经验的点

组件中必须的内容

## 论坛参考

<https://cn.vuejs.org/v2/api>

# 说一下vuex？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 90% | 5星 | Javascript | 数据存储 | 存储 |

## 通常解法

Vuex 是适用于 Vue.js 应用的状态管理库，为应用中的所有组件提供集中式的状态存储与操作，保证了所有状态以可预测的方式进行修改；

state: state 定义了应用状态的数据结构，同样可以在这里设置默认的初始状态。

actions:Actions 即是定义提交触发更改信息的描述，常见的例子有从服务端获取数据，在数据获取完成后会调用store.commit()来调用更改 Store 中的状态。可以在组件中使用dispatch来发出 Actions。

mutations: 调用 mutations 是唯一允许更新应用状态的地方。

getters: Getters 允许组件从 Store 中获取数据，譬如我们可以从 Store 中的 projectList 中筛选出已完成的项目列表

modules: modules 对象允许将单一的 Store 拆分为多个 Store 的同时保存在单一的状态树中。随着应用复杂度的增加，这种拆分能够更好地组织代码

但是vuex也有缺点就是，vuex中保存的数据是和网页的生命周期同步的，当执行页面刷新的时候vuex中所有数据都会消失复位到初始状态，所以不太适合做有分享页面的数据交互（在这种项目中vuex只适合数据的集中管理，不适合数据的存储，这种情况一般是使用路由传递参数会好一些），适合后台管理系统多一些，后台管理系统一般都是公司内部使用；

## 通用大牛级解法

Vuex 是专门针对vue开发的应用状态管理库，为应用中的所有组件提供集中式的状态存储与操作，保证了所有状态以可预测的方式进行修改；

state: state 定义了应用状态的数据结构，同样可以在这里设置默认的初始状态。

actions:Actions 即是定义提交触发更改信息的描述，常见的例子有从服务端获取数据，在数据获取完成后会调用store.commit()来调用更改 Store 中的状态。可以在组件中使用dispatch来发出 Actions。

mutations: 调用 mutations 是唯一允许更新应用状态的地方。

getters: Getters 允许组件从 Store 中获取数据，譬如我们可以从 Store 中的 projectList 中筛选出已完成的项目列表

modules: modules 对象允许将单一的 Store 拆分为多个 Store 的同时保存在单一的状态树中。随着应用复杂度的增加，这种拆分能够更好地组织代码

在项目vue中主要使用vuex需要注意的是vuex相当于一个全局的状态，所以不要在vuex保存大量的数据，只需要保存全局通用数据就可以了，还有就是当刷新页面的时候由于是js对象所以对象中的数据都会回归初始化，所以一般都需要配合本地存储一块使用，在就是vuex针对于vue的数据检测做了适配，所以在vue中使用vuex做状态管理还是比较不错的

## 解法对比及优缺点

通常解法，vuex的规则解释。通用级解法，说出了vuex的特性，优缺点，以及

。

## 项目中体现经验的点

登录判断，本地数据线上数据的同步

## 论坛参考

<https://vuex.vuejs.org/>

# 小程序的组件？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 50% | 3星 | 小程序 | 功能抽象 | 组件 |

## 通常解法

从小程序基础库版本 1.6.3 开始，小程序支持简洁的组件化编程。所有自定义组件相关特性都需要基础库版本 1.6.3 或更高。

开发者可以将页面内的功能模块抽象成自定义组件，以便在不同的页面中重复使用；也可以将复杂的页面拆分成多个低耦合的模块，有助于代码维护。自定义组件在使用时与基础组件非常相似。

在自定义组件的 js 文件中，需要使用 Component() 来注册组件，并提供组件的属性定义、内部数据和自定义方法。

组件的属性值和内部数据将被用于组件 wxml 的渲染，其中，属性值是可由组件外部传入的

组件构造器包含的属性和方法：

properties，data，methods，behaviors（mixin），created，attached，ready，moved，detached，relations等等

## 通用大牛级解法

理解小程序组件可以对比vue组件或者react组件，页面中多组件嵌套肯定会涉及跨组件传值，那么小程序的组件传值是如何实现的。Vue或者react都属于单向数据流的数据管理方式，小程序组件也是类似的。通过在组件上添加自定义属性，属性值可以是父组件的任意变量，在组建内使用properties属性来接收。例如：<test data=”{{count}}” bindtest=”func”></test>

Component({

Properties:{

data:{

Type:String,

Value:’’,

Observer:function(){}

}

}

})

这是父子组件传值，其中type限制组件接受值的格式，value可以设置该属性对应的默认值，observer类似于vue的watch，可以订阅属性值的变化，进而做相应的操作。

小程序组件由子组件向父组件传值则用的的事件派发的方式通过triggerEvent方法触发，并在组件标签上bind事件监听，执行回调函数，来实现子父传值。类似于vue的$emit, v-on组合

除了这些传值方式以外，还有全局对象挂载的方式。任何页面都可以通过getApp()得到app实例，该app实例是小程序的全局对象，拥有上帝视角，并且可以添加自定义属性，利用这点也能实现跨组件传值，不再限于父子组件之间。

## 解法对比及优缺点

* 普通解法只是小程序组件的概念和api，以及基本的使用。
* 大牛级解法对小程序组件的数据流进行了梳理，解释清楚了实际操作中会遇到的具体问题，以及解决方式，体现了丰富的经验，以及数据流概念

## 延伸及扩展问题回答参考

对比vue，react，小程序组件各自的特点，总结它们共性和不同点

## 项目中体现经验的点

深刻理解小程序组件的特点和使用方法；

## 论坛参考

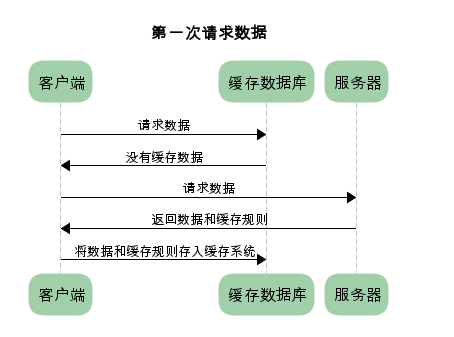
## <https://developers.weixin.qq.com/miniprogram/dev/framework/custom-component/events.html>

# 如何理解http的缓存机制？

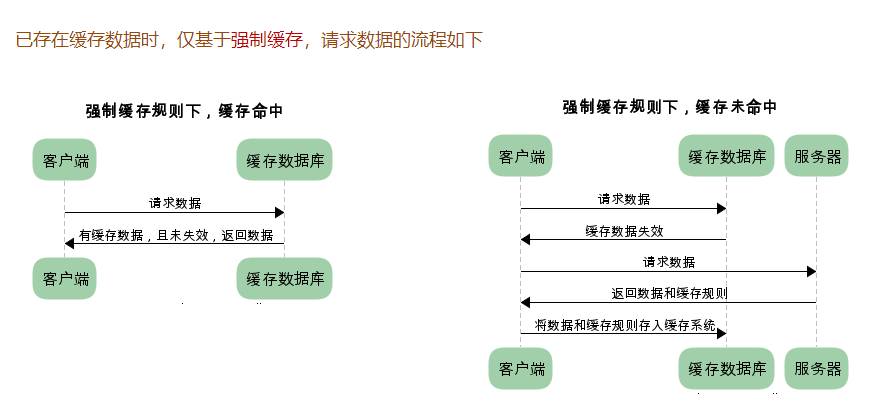
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 70% | 4星 | 面向对象 | http协议 | http缓存 |

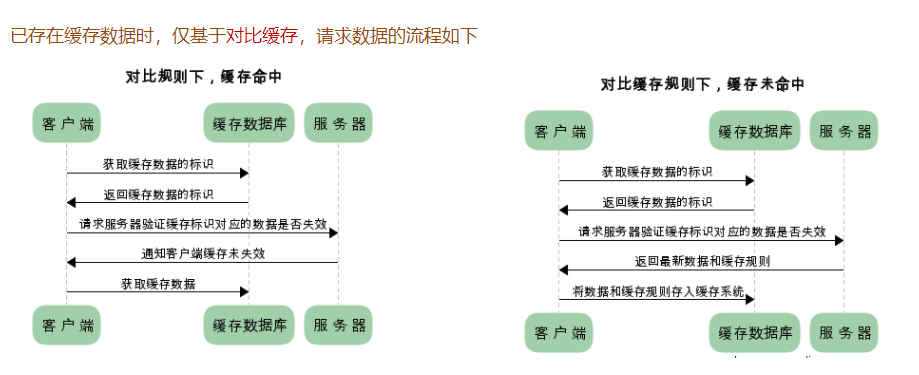
## 通常解法

在客户端第一次请求数据时，此时缓存数据库中没有对应的缓存数据，需要请求服务器，服务器返回后，将数据存储至缓存数据库中。



HTTP缓存有多种规则，根据是否需要重新向服务器发起请求来分类，我将其分为两大类(强制缓存，对比缓存)  
在详细介绍这两种规则之前，先通过时序图的方式，让大家对这两种规则有个简单了解





我们可以看到两类缓存规则的不同，强制缓存如果生效，不需要再和服务器发生交互，而对比缓存不管是否生效，都需要与服务端发生交互。  
两类缓存规则可以同时存在，强制缓存优先级高于对比缓存，也就是说，当执行强制缓存的规则时，如果缓存生效，直接使用缓存，不再执行对比缓存规则

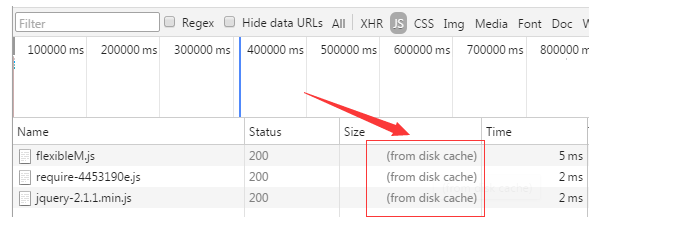
## 通用大牛级解法

从以上解法我们得知，强制缓存，在缓存数据未失效的情况下，可以直接使用缓存数据，那么浏览器是如何判断缓存数据是否失效呢？

我们知道，在没有缓存数据的时候，浏览器向服务器请求数据时，服务器会将数据和缓存规则一并返回，缓存规则信息包含在响应header中。

对于强制缓存来说，响应header中会有两个字段来标明失效规则（Expires/Cache-Control）

使用chrome的开发者工具，可以很明显的看到对于强制缓存生效时，网络请求的情况



Expires

　　Expires的值为服务端返回的到期时间，即下一次请求时，请求时间小于服务端返回的到期时间，直接使用缓存数据。

不过Expires 是HTTP 1.0的东西，现在默认浏览器均默认使用HTTP 1.1，所以它的作用基本忽略。

另一个问题是，到期时间是由服务端生成的，但是客户端时间可能跟服务端时间有误差，这就会导致缓存命中的误差。

所以HTTP 1.1 的版本，使用Cache-Control替代。

Cache-Control

Cache-Control 是最重要的规则。常见的取值有private、public、no-cache、max-age，no-store，默认为private。

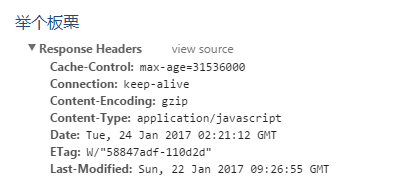
private: 客户端可以缓存

public: 客户端和代理服务器都可缓存（前端的同学，可以认为public和private是一样的）

max-age=xxx: 缓存的内容将在 xxx 秒后失效

no-cache: 需要使用对比缓存来验证缓存数据（后面介绍）

no-store: 所有内容都不会缓存，强制缓存，对比缓存都不会触发（对于前端开发来说，缓存越多越好，so...基本上和它说886）



## 解法对比及优缺点

跟家强调缓存的规则，对缓存规则有更清楚的认识！理解各个响应头字段的规则。

## 延伸及扩展问题回答参考

对比缓存

对比缓存，顾名思义，需要进行比较判断是否可以使用缓存。

浏览器第一次请求数据时，服务器会将缓存标识与数据一起返回给客户端，客户端将二者备份至缓存数据库中。

再次请求数据时，客户端将备份的缓存标识发送给服务器，服务器根据缓存标识进行判断，判断成功后，返回304状态码，通知客户端比较成功，可以使用缓存数据。

## 项目中体现经验的点

If-Modified-Since：

再次请求服务器时，通过此字段通知服务器上次请求时，服务器返回的资源最后修改时间。

服务器收到请求后发现有头If-Modified-Since 则与被请求资源的最后修改时间进行比对。

若资源的最后修改时间大于If-Modified-Since，说明资源又被改动过，则响应整片资源内容，返回状态码200；

若资源的最后修改时间小于或等于If-Modified-Since，说明资源无新修改，则响应HTTP 304，告知浏览器继续使用所保存的cache。

## 6.论坛参考

<https://blog.csdn.net/wangxiuyan0228/article/details/79457538>

<https://www.cnblogs.com/haorui/p/3583337.html>

<https://blog.csdn.net/victor_e_n_01185/article/details/72875669>

# vue-router中路由组件传参的问题？

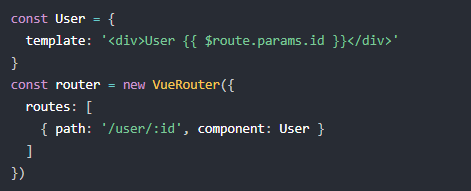
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 70% | 4星 | 面向对象 | javaScript | Vue路由 |

## 通常解法

一般我们配置vue的路由的时候，在每个路由对象中是可以配置props的，配置这个属性的目的是为了解耦路由组件；

在组件中使用 $route 会使之与其对应路由形成高度耦合，从而使组件只能在某些特定的 URL 上使用，限制了其灵活性

如下代码会产生耦合：



如下是通过props进行解耦



这样的话，就可以在任何的地方重用该组件；不用担心是在什么环境下！

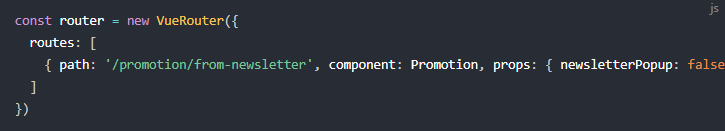
## 通用大牛级解法

对于vue路由对象中的配置，其实是提供了多种模式的。都是为了方便去使用对路由组件的传参！

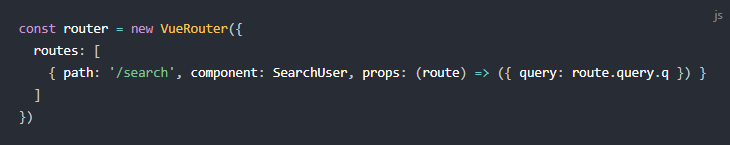
那么对于props的配置有一下三种模式：

布尔模式：如果props设置为true，route.params将会被设置为组件属性；

对象模式: 如果props是一个对象，它会被按原样设置为组件属性。如下：



函数模式：还可以创建有一个函数来返回props，这样就可以将参数转换成另一种类型，将静态的值与基于路由的值结合。



尽可能保持 props 函数为无状态的，因为它只会在路由发生变化时起作用。如果你需要状态来定义 props，请使用包装组件，这样 Vue 才可以对状态变化做出反应。

## 解法对比及优缺点

1. 对vue-router的理解更进一步
2. 掌握多种配置方式，可以针对不同的情况
3. 保持路由组件的无状态，更利于开发
4. 使$router解耦，便于重复组件的复用

## 延伸及扩展问题回答参考

实现原理：

在浏览器端大体有两种实现方式：

* + - 1. 通过hash字符串，配合onhashchange监听路由变化。进行视图的切换
      2. 通过h5新增的pushState方法修改路由的变化。但是此种方法需要配置服务端的响应。

## 项目中体现经验的点

能解耦$router和路由组件的关联，可以在任何地方复用组件

## 6.论坛参考

<https://router.vuejs.org/zh/guide/essentials/passing-props.html#%E5%B8%83%E5%B0%94%E6%A8%A1%E5%BC%8F>

<https://www.cnblogs.com/SamWeb/p/6610733.html>

<https://segmentfault.com/a/1190000011123089>

# 解释一下javaScript中的继承机制？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 95% | 3星 | Js底层原理 | javaScript | 继承模式 |

## 通常解法

javascript中继承跟java中的继承不太一样，一般通过call()和apply()两种方式完成，js中的继承是以复制的形式完成的，复制一个父对象，而不像java中直接继承父对象，还有通过原型的方式完成继承，也有弊端，总之js中的继承只是形式上的对面向对象语言的一种模仿，本质上不是继承，但用起来效果是一样的。  
至于为什么要继承：通常在一般的项目里不需要，因为应用简单，但你要用纯js做一些复杂的工具或框架系统就要用到了，比如webgis、或者js框架如jquery、ext什么的，不然一个几千行代码的框架不用继承得写几万行，甚至还无法维护。

## 通用大牛级解法

在通用解法的基础上，我们实现继承的三种方式。

**原型继承**



这种原型继承的特点：既继承了父类的模板，又继承了父类的原型对象。优点是继承了父类的模板，又继承了父类的原型对象，缺点就是父类实例传参，不是子类实例化传参，不符合常规语言的写法。

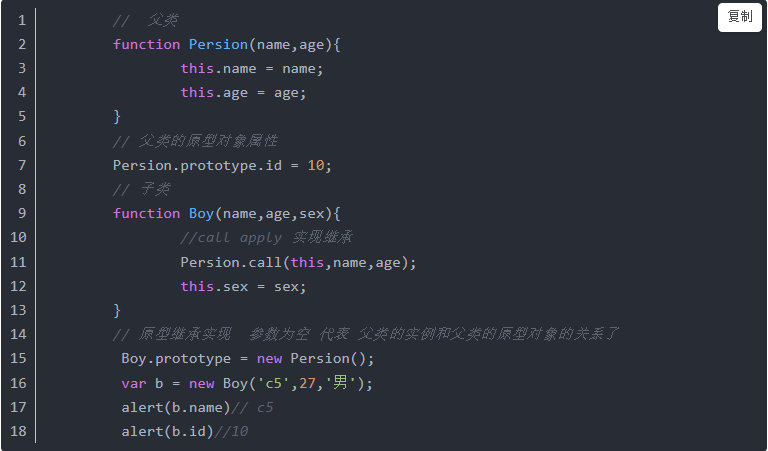
**类继承（借用构造函数的方式继承）**



这种原型继承的特点：继承了父类的模板，不继承了父类的原型对象。优点是方便了子类实例传参，缺点就是不继承了父类的原型对象

因而就可以将其占用的内存空间回收回来。这样，当垃圾收集器下次再运行时，它就会释放那些引用次数为零的值所占用的内存。

**混合继承（原型继承和类继承）**



这种原型继承的特点：既继承了父类的模板，又继承了父类的原型对象。优点方便了子类实例传参，缺点就是Boy.pertotype = new Persion()  函数又实例一次，函数内部变量又重复实例一次，大程序时候会很好性能。

## 解法对比及优缺点

对比以上解法，我们发现在通用解法中我们只说到JavaScript的继承本质和为什么要使用继承。而在大牛级解法中我们举出了三个常用继承的模式并加以代码说明，这样一来回答的更加具体和形象

## 延伸及扩展问题回答参考

上面是ES5的继承机制，对应即将广泛应用的ES6,我们也要了解它的新继承机制

Class之间可以通过extends关键字实现继承，这比ES5的通过修改原型链实现继承，要清晰和方便很多。

class ColorPoint extends Point {}

上面代码定义了一个ColorPoint类， 该类通过extends关键字， 继承了Point类的所有属性和方法。 但是由于没有部署任何代码， 所以这两个类完全一样， 等于复制了一个Point类。 下面， 我们在ColorPoint内部加上代码。

class ColorPoint extends Point {

constructor(x, y, color) {

super(x, y); // 调用父类的 constructor(x, y)

this.color = color;

}

toString() {

return this.color + ' ' + super.toString(); // 调用父类的 toString()

}

}

上面代码中， constructor方法和toString方法之中， 都出现了super关键字， 它在这里表示父类的构造函数， 用来新建父类的this对象。  
子类必须在constructor方法中调用super方法， 否则新建实例时会报错。 这是因为子类没有自己的this对象， 而是继承父类的this对象， 然后对其进行加工。 如果不调用super方法， 子类就得不到this对象。

class Point { /\* ... \*/ }

class ColorPoint extends Point {

constructor() {}

}

let cp = new ColorPoint(); // ReferenceError

上面代码中， ColorPoint继承了父类Point， 但是它的构造函数没有调用super方法， 导致新建实例时报错。  
ES5 的继承， 实质是先创造子类的实例对象this， 然后再将父类的方法添加到this上面（ Parent.apply(this)）。 ES6 的继承机制完全不同， 实质是先创造父类的实例对象this（ 所以必须先调用super方法）， 然后再用子类的构造函数修改this。  
如果子类没有定义constructor方法， 这个方法会被默认添加， 代码如下。 也就是说， 不管有没有显式定义， 任何一个子类都有constructor方法。

constructor(...args) {

super(...args);

}

另一个需要注意的地方是， 在子类的构造函数中， 只有调用super之后， 才可以使用this关键字， 否则会报错。 这是因为子类实例的构建， 是基于对父类实例加工， 只有super方法才能返回父类实例。

class Point {

constructor(x, y) {

this.x = x;

this.y = y;

}

}

class ColorPoint extends Point {

constructor(x, y, color) {

this.color = color; // ReferenceError

super(x, y);

this.color = color; // 正确

}

}

上面代码中， 子类的constructor方法没有调用super之前， 就使用this关键字， 结果报错， 而放在super方法之后就是正确的。  
下面是生成子类实例的代码。

let cp = new ColorPoint(25, 8, 'green');

cp instanceof ColorPoint // true

cp instanceof Point // true

上面代码中， 实例对象cp同时是ColorPoint和Point两个类的实例， 这与 ES5 的行为完全一致。

## 项目中体现经验的点

深刻理解JavaScript的继承机制，利用继承机制来优化和精简外面的代码；

## 论坛参考

<https://blog.csdn.net/caijixin/article/details/78295676>

<https://blog.csdn.net/qq_30100043/article/details/53542531>

# 解释一下经常使用的vue数据管理方案？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 90% | 3星 | 框架 | Vue | 数据管理 |

## 通常解法

vue的每个组件中都会维护一个自身数据管理属性：data和一个父组件传过来的数据属性：props，我们只要合理利用data和props就可以解决大部分应用场景，比如父子组件间通信和兄弟组件间通信。再复杂一点的场景我们可以用到Bus和本地存储技术，实现跨组件间的数据通信。

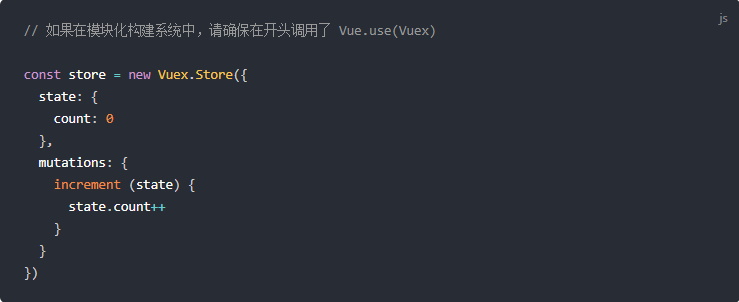
## 通用大牛级解法

在通用解法的基础上，介绍数据管理工具：Vuex

Vuex是一个专为 Vue.js 应用程序开发的状态管理模式。它采用集中式存储管理应用的所有组件的状态，并以相应的规则保证状态以一种可预测的方式发生变化。

每一个 Vuex 应用的核心就是 store（仓库）。“store”基本上就是一个容器，它包含着你的应用中大部分的状态 (state)。Vuex 和单纯的全局对象有以下两点不同：

* Vuex 的状态存储是响应式的。当 Vue 组件从 store 中读取状态的时候，若 store 中的状态发生变化，那么相应的组件也会相应地得到高效更新。
* 你不能直接改变 store 中的状态。改变 store 中的状态的唯一途径就是显式地提交 (commit) mutation。这样使得我们可以方便地跟踪每一个状态的变化，从而让我们能够实现一些工具帮助我们更好地了解我们的应用。



现在，你可以通过 store.state 来获取状态对象，以及通过 store.commit 方法触发状态变更：



我们通过提交 mutation 的方式，而非直接改变 store.state.count，是因为我们想要更明确地追踪到状态的变化。这个简单的约定能够让你的意图更加明显，这样你在阅读代码的时候能更容易地解读应用内部的状态改变。此外，这样也让我们有机会去实现一些能记录每次状态改变，保存状态快照的调试工具。有了它，我们甚至可以实现如时间穿梭般的调试体验。

由于 store 中的状态是响应式的，在组件中调用 store 中的状态简单到仅需要在计算属性中返回即可。触发变化也仅仅是在组件的 methods 中提交 mutation。

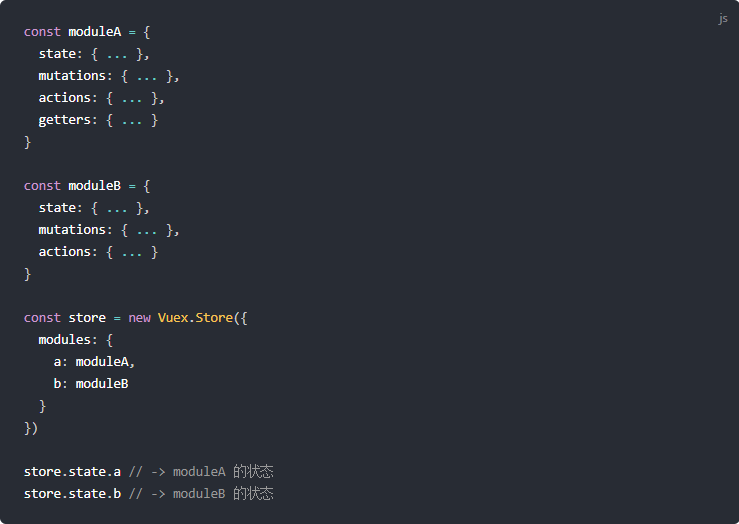
## 解法对比及优缺点

对比以上解法，我们发现在通用解法中我们只说到Vue自身的状态管理和BUS做组件间的通信，而在大牛级解法中我们给出了最新的类flux架构的vuex。Vuex既可以帮我们解决数据管理，并以相应的规则保证状态以一种可预测的方式发生变化来实现组件间的通信。这种解法让我们在回答问题时更有层次，我们的技术深度也能够更好的体现出来。

## 延伸及扩展问题回答参考

由于使用单一状态树，应用的所有状态会集中到一个比较大的对象。当应用变得非常复杂时，store 对象就有可能变得相当臃肿。

为了解决以上问题，Vuex 允许我们将 store 分割成模块（module）。每个模块拥有自己的 state、mutation、action、getter、甚至是嵌套子模块——从上至下进行同样方式的分割：



## 项目中体现经验的点

深刻理解vue的数据管理机制，在项目中管理好数据会帮助我们写出更加简洁高效的代码。

## 论坛参考

https://vuex.vuejs.org/zh/

<https://vuex.vuejs.org/zh/guide/modules.html>

[https://segmentfault.com/a/1190000010264128](https://vuex.vuejs.org/zh/guide/modules.html)

# Js的运行原理？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 95% | 3星 | Es6 | 异步 | Es6异步操作 |

## 通常解法

js是单线程运行的，分为异步任务和同步任务

## 通用大牛级解法

JavaScript引擎是单线程运行的,浏览器无论在什么时候都只且只有一个线程在运行JavaScript程序.浏览器的内核是多线程的，它们在内核制控下相互配合以保持同步，一个浏览器至少实现三个常驻线程：javascript引擎线程，GUI渲染线程，浏览器事件触发线程。这些异步线程都会产生不同的异步的事件.

1. javascript引擎是基于事件驱动单线程执行的，JS引擎一直等待着任务队列中任务的到来，然后加以处理，浏览器无论什么时候都只有一个JS线程在运行JS程序。
2. GUI渲染线程负责渲染浏览器界面，当界面需要重绘（Repaint）或由于某种操作引发回流(reflow)时,该线程就会执行。但需要注意 GUI渲染线程与JS引擎是互斥的，当JS引擎执行时GUI线程会被挂起，GUI更新会被保存在一个队列中等到JS引擎空闲时立即被执行。
3. 事件触发线程，当一个事件被触发时该线程会把事件添加到待处理队列的队尾，等待JS引擎的处理。这些事件可来自JavaScript引擎当前执行的代码块如setTimeOut、也可来自浏览器内核的其他线程如鼠标点击、AJAX异步请求等，但由于JS的单线程关系所有这些事件都得排队等待JS引擎处理。（当线程中没有执行任何同步代码的前提下才会执行异步代码）

当程序启动时, 一个进程被创建，同时也运行一个线程, 即为主线程，js的运行机制为单线程

程序中跑两个线程，一个负责程序本身的运行，作为主线程； 另一个负责主线程与其他线程的的通信，被称为“Event Loop 线程" 。每当遇到异步任务，交给 EventLoop 线程，然后自己往后运行，等到主线程运行完后，再去 EventLoop 线程拿结果。

1）所有任务都在主线程上执行，形成一个执行栈（execution context stack）。

2）主线程之外，还存在一个"任务队列"（task queue）。系统把异步任务放到"任务队列"之中，然后继续执行后续的任务。

3）一旦"执行栈"中的所有任务执行完毕，系统就会读取"任务队列"。如果这个时候，异步任务已经结束了等待状态，就会从"任务队列"进入执行栈，恢复执行。

4）主线程不断重复上面的第三步。

"回调函数"（callback），就是那些会被主线程挂起来的代码。异步任务必须指定回调函数，当异步任务从"任务队列"回到执行栈，回调函数就会执行。"任务队列"是一个先进先出的数据结构，排在前面的事件，优先返回主线程。主线程的读取过程基本上是自动的，只要执行栈一清空，"任务队列"上第一位的事件就自动返回主线程。

主线程从"任务队列"中读取事件，这个过程是循环不断的，所以整个的这种运行机制又称为**Event Loop。**

**从主线程的角度看，一个异步过程包括下面两个要素：**

发起函数(或叫注册函数)A

回调函数callbackFn

它们都是在主线程上调用的，其中注册函数用来发起异步过程，回调函数用来处理结果。

**异步进程有:**

类似onclick等，由浏览器内核的DOM binding模块处理，事件触发时，回调函数添加到任务队列中；

setTimeout等，由浏览器内核的Timer模块处理，时间到达时，回调函数添加到任务队列中；

Ajax，由浏览器内核的Network模块处理，网络请求返回后，添加到任务队列中。

例如setTimeout(fn, 1000)，其中的setTimeout就是异步过程的发起函数，fn是回调函数。用一句话概括：工作线程将消息放到消息队列，主线程通过事件循环过程去取消息。

消息队列：消息队列是一个先进先出的队列，它里面存放着各种消息。

事件循环：事件循环是指主线程重复从消息队列中取消息、执行的过程。

**流程如下:**

1. 主线程读取js代码, 形成相应的堆和执行栈, 执行同步任务
2. 当主线程遇到异步任务,，指定给异步进程处理, 同时继续执行同步任务
3. 当异步进程处理完毕后， 将相应的异步任务推入到任务队列首部
4. 主线程任务处理完毕后,，查询任务队列，则取出一个任务队列推入到主线程的执行栈

重复执行第2、3、4步，这就称为事件循环

## 解法对比及优缺点

**综上所述**：第一种只是关键字，而第二种说出其中的正真的执行原理

## 项目中体现经验的点

程序的执行顺序

## 论坛参考

# 说一下前端安全？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 90% | 5星 | Javascript | 数据存储 | 存储 |

## 通常解法

恶意攻击者往Web页面里注入恶意Script代码，用户浏览这些网页时，就会执行其中的恶意代码，可对用户进行盗取cookie信息、会话劫持等各种攻击.；

## 通用大牛级解法

1. **1，XSS（Cross Site Scripting，跨站脚本攻击）**
2. 这是前端最常见的攻击方式，很多大型网站（如 Facebook）都被 XSS 攻击过。
3. 举一个例子，我在一个博客网站正常发表一篇文章，输入汉字、英文和图片，完全没有问题。但是如果我写的是恶意的 JS 脚本，例如获取到document.cookie然后传输到自己的服务器上，那我这篇博客的每一次浏览都会执行这个脚本，都会把访客 cookie 中的信息偷偷传递到我的服务器上来。
4. 其实原理上就是黑客通过某种方式（发布文章、发布评论等）将一段特定的 JS 代码隐蔽地输入进去。然后别人再看这篇文章或者评论时，之前注入的这段 JS 代码就执行了。JS 代码一旦执行，那可就不受控制了，因为它跟网页原有的 JS 有同样的权限，例如可以获取 server 端数据、可以获取 cookie 等。于是，攻击就这样发生了。
5. XSS的危害
6. XSS 的危害相当大，如果页面可以随意执行别人不安全的 JS 代码，轻则会让页面错乱、功能缺失，重则会造成用户的信息泄露。
7. 比如早些年社交网站经常爆出 XSS 蠕虫，通过发布的文章内插入 JS，用户访问了感染不安全 JS 注入的文章，会自动重新发布新的文章，这样的文章会通过推荐系统进入到每个用户的文章列表面前，很快就会造成大规模的感染。
8. 还有利用获取 cookie 的方式，将 cookie 传入入侵者的服务器上，入侵者就可以模拟 cookie 登录网站，对用户的信息进行篡改。
9. XSS的预防
10. 那么如何预防 XSS 攻击呢？—— 最根本的方式，就是对用户输入的内容进行验证和替换，需要替换的字符有：
11. & 替换为：&amp;
12. < 替换为：&lt;
13. > 替换为：&gt;
14. ” 替换为：&quot;
15. ‘ 替换为：&#x27;
16. / 替换为：&#x2f;
17. 替换了这些字符之后，黑客输入的攻击代码就会失效，XSS 攻击将不会轻易发生。
18. 除此之外，还可以通过对 cookie 进行较强的控制，比如对敏感的 cookie 增加http-only限制，让 JS 获取不到 cookie 的内容。
19. **2，CSRF（Cross-site request forgery，跨站请求伪造）**
20. CSRF 是借用了当前操作者的权限来偷偷地完成某个操作，而不是拿到用户的信息。
21. 例如，一个支付类网站，给他人转账的接口是<http://buy.com/pay?touid=999&money=100>，而这个接口在使用时没有任何密码或者 token 的验证，只要打开访问就直接给他人转账。一个用户已经登录了[http://buy.com](http://buy.com/)，在选择商品时，突然收到一封邮件，而这封邮件正文有这么一行代码<img src="<http://buy.com/pay?touid=999&money=100>"/>，他访问了邮件之后，其实就已经完成了购买。
22. CSRF 的发生其实是借助了一个 cookie 的特性。我们知道，登录了[http://buy.com](http://buy.com/)之后，cookie 就会有登录过的标记了，此时请求<http://buy.com/pay?touid=999&money=100>是会带着 cookie 的，因此 server 端就知道已经登录了。而如果在[http://buy.com](http://buy.com/)去请求其他域名的 API 例如<http://abc.com/api>时，是不会带 cookie 的，这是浏览器的同源策略的限制。但是 —— 此时在其他域名的页面中，请求<http://buy.com/pay?touid=999&money=100>，会带着[buy.com](http://buy.com/)的 cookie ，这是发生 CSRF 攻击的理论基础。
23. 预防 CSRF 就是加入各个层级的权限验证，例如现在的购物网站，只要涉及现金交易，肯定要输入密码或者指纹才行。除此之外，敏感的接口使用POST请求而不是GET也是很重要的。

## 3.解法对比及优缺点

通常解法，前端常见的安全问题。通用级解法，说出了如何模拟攻击，如何防御攻击

。

## 4.项目中体现经验的点

登录权限控制，富文本内容发布

## 5.论坛参考

# Promise对象的理解？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 50% | 3星 | 小程序 | 功能抽象 | 组件 |

## 通常解法

Promise 是异步编程的一种解决方案，比传统的解决方案——回调函数和事件——更合理和更强大。它由社区最早提出和实现，ES6 将其写进了语言标准，统一了用法，原生提供了Promise对象。

所谓Promise，简单说就是一个容器，里面保存着某个未来才会结束的事件（通常是一个异步操作）的结果。从语法上说，Promise 是一个对象，从它可以获取异步操作的消息。Promise 提供统一的 API，各种异步操作都可以用同样的方法进行处理。

基本用法：

const promise = new Promise(function(resolve, reject) {

// ... some code

if (/\* 异步操作成功 \*/){

resolve(value);

} else {

reject(error);

}

});

提供的方法

Promise.prototype.then()

Promise.prototype.catch()

Promise.prototype.finally()

Promise.all()

Promise.race()

## 通用大牛级解法

Promise对象的特点：

1. 对象的状态不受外界影响。Promise对象代表一个异步操作，有三种状态：pending（进行中）、fulfilled（已成功）和rejected（已失败）。
2. 一旦状态改变，就不会再变，任何时候都可以得到这个结果。Promise对象的状态改变，只有两种可能：从pending变为fulfilled和从pending变为rejected。只要这两种情况发生，状态就凝固了，不会再变了，会一直保持这个结果，这时就称为 resolved（已定型）。

有了Promise对象，就可以将异步操作以同步操作的流程表达出来，避免了层层嵌套的回调函数。此外，Promise对象提供统一的接口，使得控制异步操作更加容易。

Promise也有一些缺点。首先，无法取消Promise，一旦新建它就会立即执行，无法中途取消。其次，如果不设置回调函数，Promise内部抛出的错误，不会反应到外部。第三，当处于pending状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）。

如果某些事件不断地反复发生，一般来说，使用 Stream 模式是比部署Promise更好的选择。

除了promise也可以考虑es6的generator和async函数

## 解法对比及优缺点

* 普通解法只是promise对象的概念，以及基本的使用。
* 大牛级解法对promise的特点进行了解释，也总结了promise的缺点，并提供了更多的选项，体现了经验。

## 延伸及扩展问题回答参考

JavaScript的异步编程总结；

Ajax的异步原理是什么

## 项目中体现经验的点

深刻理解promise的特点和使用方法；

## 论坛参考

https://developers.weixin.qq.com/miniprogram/dev/framework/custom-component/events.html

# 解释一下移动端多倍屏1px像素问题：陈蔓杰

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 90% | 3星 | 布局 | 页面布局 | 多倍屏适配 |

## 通常解法

先介绍下多倍屏：

多倍屏是指1个逻辑像素对应多个物理像素。在2倍屏下，1个逻辑像素就对应2个物理像素；在3倍屏下，1个逻辑像素就对应3个物理像素。正如我们的标准设计稿尺寸是750\*1334(也就是iphone6、7、8能够显示的像素)，虽然我们的iphone6、7、8的屏幕尺寸是375\*667，但屏幕是2倍屏，所以可以显示的实际像素是750\*1334。常见的2倍屏是iphone6、7、8，常见的3倍屏是iphone6plus、7plus、8plus。

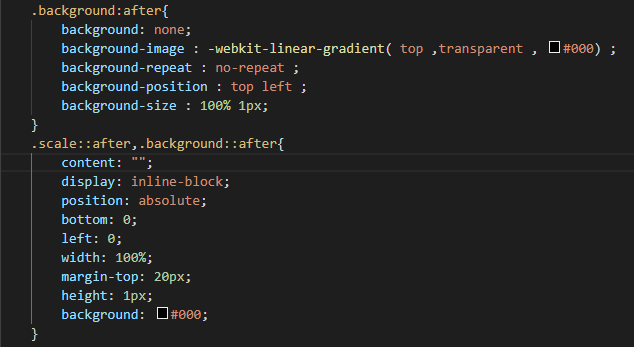
1px像素问题：

1px像素问题来了，在多倍屏下逻辑上的1像素显示的多个物理像素。那么解决方案也就来了。如在两倍屏下，1个CSS像素对应2个物理像素， 那么我们只需要想办法把border的宽度变为0.5px, 那么展现就是1个物理像素了，也就能解决1px像素问题了。

效果预览：



线性渐变：



## 通用大牛级解法

使用伪元素绘制border + 根据多倍屏进行缩放 和 设置border宽度为小数（仅ios支持）



## 解法对比及优缺点

|  |  |  |  |
| --- | --- | --- | --- |
| 不同类型的优缺点 | 背景渐变 | border的宽度为小数 | 伪元素+缩放 |
| 优点 | 使用简单，可以设置多条边 | 简单方便 | 可以适配任意多倍屏 |
| 缺点 | 只能解决两倍屏 | Ios支持，android不支持 | 代码稍微繁琐 |

## 延伸及扩展问题回答参考

还有多种其他解决1px像素的方案，常见的如：viewport缩放+rem布局，border-image，box-shadow等，这些方案都会有些自己的局限性，可以参考下面的链接做进一步研究

## 项目中体现经验的点

为了更好的还原设计稿，我们需要兼容各种多倍屏，这样可以使我们的页面看起来更加美观、优雅

## 论坛参考

<https://www.jianshu.com/p/7e63f5a32636>

<http://www.runoob.com/css3/css3-gradients.html>

https://blog.csdn.net/caochenyu0/article/details/53813074

• InnoDB通过使用MVCC(多版本并发控制：读不会阻塞写，写也不会阻塞读)来获得高并发性，并且实现了SQL标准的4种隔离级别(默认为REPEATABLE级别)；

• InnoDB还提供了插入缓冲(insert buffer)、二次写(double write)、自适应哈希索引(adaptive hash index)、预读(read ahead)等高性能和高可用的功能；

• InnoDB采用了聚集(clustered)的方式来存储表中的数据，每张标的存储都按主键的顺序存放(如果没有显式的在建表时指定主键，InnoDB会为每一行生成一个6字节的ROWID，并以此作为主键)；

• InnoDB表会有三个隐藏字段：除了上面提到了6字节的DB\_ROW\_ID外，还有6字节的DB\_TX\_ID(事务ID)和7字节的DB\_ROLL\_PTR(指向对应回滚段的地址)。这个可以通过innodb monitor看到；

MyISAM存储引擎: 是MySQL官方提供的存储引擎，主要面向OLAP(Online Analytical Processing,在线分析处理)方面的应用。

特点：

• 不支持事务，支持表所和全文索引。操作速度快；

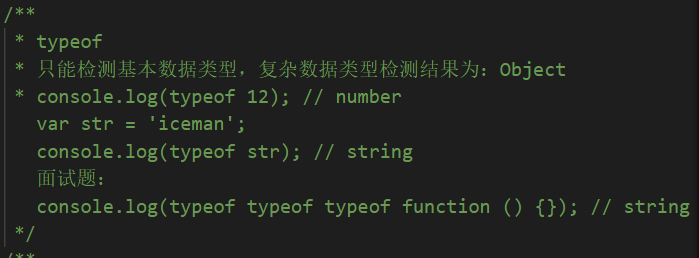
• MyISAM存储引擎表由MYD和MYI组成，MYD用来存放数据文件，MYI用来存放索引文件。MySQL数据库只缓存其索引文件，数据文件的缓存交给操作系统本身来完成；

MySQL5.0版本开始，MyISAM默认支持256T的单表数据；

# 解释一下经常使用的数据类型检测：陈蔓杰

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 90% | 3星 | 框架 | 类型检测 | 检测数据类型 |

## 通常解法

使用typeof检测数据类型，首先返回的都是一个字符串，其次字符串中包含了对应的数据类型，例如："number"、"string"、"boolean"、"undefined"、"function"、"object"

## 通用大牛级解法

在通用解法的基础上，介绍数据类型检测

**① instanceof**

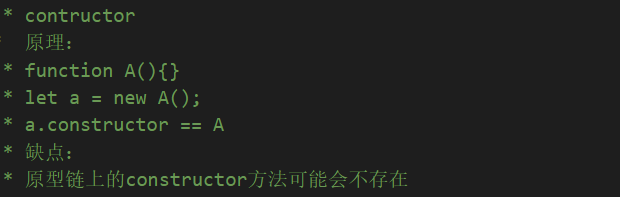
只要在当前实例的原型链上，用instanceof检测出来的结果都是true，所以在类的原型继承中，最后检测出来的结果未必是正确的

② **constructor**

constructor即构造函数，作用和instanceof非常的相似。

constructor可以处理基本数据类型的检测

constructor检测Object和instanceof不一样，一般情况下是检测不了的 constructor的局限性：我们可以把类的原型进行重写，在重写的过程中，很有可能把之前的constructor给覆盖了，这样检测出来的结果就是不准确的。对于特殊的数据类型null和undefined，它们的所属类型是Null和Undefined，但是浏览器把这两个类保护起来了，不允许在外面访问使用

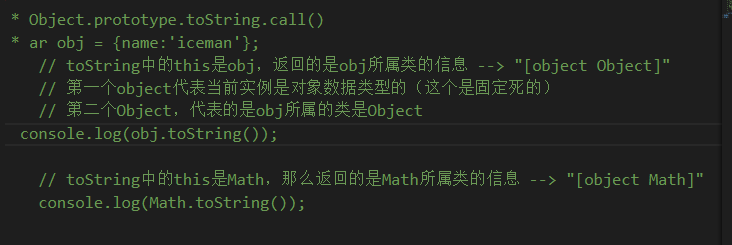


③ **Object.prototype.toStrong.call()**

Object.prototype.toStrong.call()是检测数据类型最准确最常用的方式，起原理为：

先获取Object原型上的toString方法，让方法执行，并且改变方法中的this关键字的指向；

Object.prototype.toString 它的作用是返回当前方法的执行主体（方法中this）所属类的详细信息

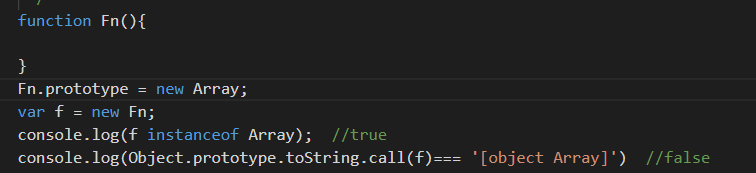


## 解法对比及优缺点

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 不同类型的优缺点 | typeof | instanceof | constructor | Object.prototype.toString.call |
| 优点 | 使用简单 | 能检测出引用类型 | 基本能检测所有的类型（null和undefined） | 检测出所有的类型 |
| 缺点 | 只能检测出基本类型（除null） | 不能检测出基本类型且不能跨iframe | constructor易被修改，也不能跨iframe | IE6下，undefined和null均为Object |

## 延伸及扩展问题回答参考

检测原型继承的情况：



## 项目中体现经验的点

深刻理解数据的基本类型检测，在项目中当我们不确定函数中的类型是，我们可以利用数据类型来检测函数中的类型，从而方便我们更好的完成任务

## 论坛参考

[https://www.talkingcoder.com/article/6333557442705696719](https://www.talkingcoder.com/article/6333557442705696719" \t "https://www.jianshu.com/p/_blank)

<https://www.jianshu.com/p/75057391ad51>

https://blog.csdn.net/lhjuejiang/article/details/79623973

# 什么是MVVM？：刘于项目经理

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 95% | 3星 | mvvm | Mvvm | 什么是MVVM？ |

## 通常解法

1. MVVM 是 Model-View-ViewModel 的缩写。mvvm 是一种设计思想。

## 通用大牛级解法

1. MVVM 是 Model-View-ViewModel 的缩写。mvvm 是一种设计思想。Model 层代表数据模型，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 是一个同步 View 和 Model 的对象。
2. 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。
3. 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。
4. mvc 和 mvvm 其实区别并不大。都是一种设计思想。主要就是 mvc 中 Controller 演变成 mvvm 中的 viewModel。mvvm 主要解决了 mvc 中大量的 DOM 操作使页面渲染性能降低，加载速度变慢，影响用户体验。和当 Model 频繁发生变化，开发者需要主动更新到 View 。

5.Mvvm:

Model：数据模型

View：带特殊属性的 html 模板

ViewModel：依靠 Directive，修改数据 & 自动渲染

6.MVVM的实现主要是三个核心点：

响应式：vue如何监听data的属性变化

模板解析：vue的模板是如何被解析的

渲染：vue模板是如何被渲染成HTML的

## 解法对比及优缺点

**综上所述**：第一种只是关键字，而第二种说出其中的正真的执行原理

## 项目中体现经验的点

对vue中mvvm的掌握程度

## 论坛参考

# Vue的响应原理：陈蔓杰项目经理

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 95% | 3星 | Vue | vue | Vue的响应原理 |

## 通常解法

Vue的数据响应主要是依赖了Object.defineProperty()，响应式系统依赖于三个重要的类：Dep 类、Watcher 类、Observer 类。Dep 类作为发布者的角色，Watcher 类作为订阅者的角色，Observer 类则是连接发布者和订阅者的纽带，决定订阅和发布的时机。在 Observer 类中，为 data 的每个属性都实例化一个 Dep 类，即发布者。并且在取值时让订阅者订阅，在赋值时发布者发布通知，让订阅者做出各自的响应。Watcher 类在构造函数中执行了一个方法，将自己赋值，并且执行了取值操作，这样就成功的完成了订阅操作。一旦数据发生变化，即有了赋值操作，发布者就会发送通知，订阅者就会执行自己的 update 方法来响应这次数据变化

## 通用大牛级解法

Vue 最独特的特性之一，是其非侵入性的响应式系统。数据模型仅仅是普通的 JavaScript 对象。而当你修改它们时，视图会进行更新。这使得状态管理非常简单直接，不过理解其工作原理同样重要，这样你可以回避一些常见的问题。

当你把一个普通的 JavaScript 对象传给 Vue 实例的 data 选项，Vue 将遍历此对象所有的属性，并使用 [Object.defineProperty](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty" \t "https://cn.vuejs.org/v2/guide/_blank) 把这些属性全部转为 [getter/setter](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Working_with_Objects" \l "%E5%AE%9A%E4%B9%89_getters_%E4%B8%8E_setters" \t "https://cn.vuejs.org/v2/guide/_blank)。Object.defineProperty 是 ES5 中一个无法 shim 的特性，这也就是为什么 Vue 不支持 IE8 以及更低版本浏览器。

这些 getter/setter 对用户来说是不可见的，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。这里需要注意的问题是浏览器控制台在打印数据对象时 getter/setter 的格式化并不同，所以你可能需要安装 [vue-devtools](https://github.com/vuejs/vue-devtools" \t "https://cn.vuejs.org/v2/guide/_blank) 来获取更加友好的检查接口。每个组件实例都有相应的 watcher 实例对象，它会在组件渲染的过程中把属性记录为依赖，之后当依赖项的 setter 被调用时，会通知 watcher 重新计算，从而致使它关联的组件得以更新。

受现代 JavaScript 的限制 (而且 Object.observe 也已经被废弃)，Vue 不能检测到对象属性的添加或删除。由于 Vue 会在初始化实例时对属性执行 getter/setter 转化过程，所以属性必须在 data 对象上存在才能让 Vue 转换它，这样才能让它是响应的。

由于 Vue 不允许动态添加根级响应式属性，所以你必须在初始化实例前声明根级响应式属性，哪怕只是一个空值，如果你未在 data 选项中声明 message，Vue 将警告你渲染函数正在试图访问的属性不存在。这样的限制在背后是有其技术原因的，它消除了在依赖项跟踪系统中的一类边界情况，也使 Vue 实例在类型检查系统的帮助下运行的更高效。而且在代码可维护性方面也有一点重要的考虑：data 对象就像组件状态的概要，提前声明所有的响应式属性，可以让组件代码在以后重新阅读或其他开发人员阅读时更易于被理解。

可能你还没有注意到，Vue 异步执行 DOM 更新。只要观察到数据变化，Vue 将开启一个队列，并缓冲在同一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。这种在缓冲时去除重复数据对于避免不必要的计算和 DOM 操作上非常重要。然后，在下一个的事件循环“tick”中，Vue 刷新队列并执行实际 (已去重的) 工作。Vue 在内部尝试对异步队列使用原生的 Promise.then 和 MessageChannel，如果执行环境不支持，会采用 setTimeout(fn, 0) 代替。

例如，当你设置 vm.someData = 'new value' ，该组件不会立即重新渲染。当刷新队列时，组件会在事件循环队列清空时的下一个“tick”更新。多数情况我们不需要关心这个过程，但是如果你想在 DOM 状态更新后做点什么，这就可能会有些棘手。虽然 Vue.js 通常鼓励开发人员沿着“数据驱动”的方式思考，避免直接接触 DOM，但是有时我们确实要这么做。为了在数据变化之后等待 Vue 完成更新 DOM ，可以在数据变化之后立即使用 Vue.nextTick(callback) 。这样回调函数在 DOM 更新完成后就会调用。

## 解法对比及优缺点

**综上所述**：第一种只是关键字，而第二种说出其中的正真的执行原理

## 项目中体现经验的点

事件的构建机制原理，以及如何运行

## 论坛参考

# 高阶组件原理：陈蔓杰项目经理

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 95% | 3星 | React | 高阶组件 | 高阶组件 |

## 通常解法

接收函数作为输入，或者输出另一个函数的一类函数，被称作高阶函数

## 通用大牛级解法

**高阶组件**通过包裹（wrapped）被传入的React组件，经过一系列处理，最终返回一个相对增强（enhanced）的React组件，供其他组件调用。

高阶组件是react应用中很重要的一部分，最大的特点就是重用组件逻辑。它并不是由

React API定义出来的功能，而是由React的组合特性衍生出来的一种设计模式。

2.高阶组件可以看做是装饰器模式(Decorator Pattern)在React的实现。即允许向一个现有的对象添加新的功能，同时又不改变其结构，属于包装模式(Wrapper Pattern)的一种

3.ES7中添加了一个decorator的属性，使用@符表示，可以更精简的书写

**两种形式：**

A、属性代理

引入里我们写的最简单的形式，就是属性代理(Props Proxy)的形式。

a.操作props  
最直观的就是接受到props，我们可以做任何读取，编辑，删除的很多自定义操作。包括hoc中定义的自定义事件，都可以通过props再传下去

b.refs获取组件实例  
当我们包装Usual的时候，想获取到它的实例怎么办，可以通过引用(ref),在Usual组件挂载的时候，会执行ref的回调函数，在hoc中取到组件的实例。

c.抽离state

这里不是通过ref获取state， 而是通过 { props, 回调函数 } 传递给wrappedComponent组件，通过回调函数获取state。这里用的比较多的就是react处理表单的时候。通常react在处理表单的时候，一般使用的是受控组件（[文档](https://link.juejin.im/?target=https%3A%2F%2Ffacebook.github.io%2Freact%2Fdocs%2Fforms.html%23controlled-components" \t "_blank)），即把input都做成受控的，改变value的时候，用onChange事件同步到state中。当然这种操作通过Container组件也可以做到，具体的区别放到后面去比较

B、反向继承

. 反向继承(Inheritance Inversion)，简称II，跟属性代理的方式不同的是，II采用通过 去继承WrappedComponent，本来是一种嵌套的关系，结果II返回的组件却继承了WrappedComponent，这看起来是一种反转的关系。  
通过继承WrappedComponent，除了一些静态方法，包括生命周期，state，各种function，我们都可以得到

**我的应用场景**

* 通常我会通过高阶组件去优化之前老项目写的不好的地方，比如两个页面UI几乎一样，功能几乎相同，仅仅几个操作不太一样，却写了两个耦合很多的页面级组件。当我去维护它的时候，由于它的耦合性过多，经常会添加一个功能(这两个组件都要添加)，我要去改完第一个的时候，还要改第二个。而且有时候由于我的记性不好，会忘掉第二个... 就会出现bug再返工。更重要的是由于个人比较懒，不想去重构这部分的代码，因为东西太多了，花费太多时间。所以加新功能的时候，我会写一个高阶组件，往HOC里添加方法，把那两个组件包装一下，也就是属性代理。这样新代码就不会再出现耦合，旧的逻辑并不会改变，说不定哪天心情好就会抽离一部分功能到HOC里，直到理想的状态。
* 另一种情况就是之前写过一个组件A，做完上线，之后产品加了一个新需求，很奇怪要做的组件B跟A几乎一模一样，但稍微有区别。那我可能就通过II的方式去继承之前的组件A，比如它在didMount去fetch请求，需要的数据是一样的。不同的地方我就会放到HOC里，存储新的state这样，再通过劫持渲染，把不同的地方，添加的地方进行处理。但其实这算Hack的一种方式，能快速解决问题，也反映了组件设计规划之初有所不足(原因比较多)。
* Container解决不了的时候甚至不太优雅的时候。其实大部分时候包一层Container组件也能做到差不多的效果，比如操作props，渲染劫持。但其实还是有很大区别的。比如我们现在有两个功能的container，添加样式和添加处理函数的，对Usual进行包装

**注意点(约束)**

* 最重要的原则就是，注意高阶组件不会修改子组件，也不拷贝子组件的行为。高阶组件只是通过组合的方式将子组件包装在容器组件中，是一个无副作用的纯函数
* 要给hoc添加class名，便于debugger。我上面的好多栗子组件都没写class 名，请不要学我，因为我实在想不出叫什么名了... 当我们在chrome里应用React-Developer-Tools的时候，组件结构可以一目了然，所以DisplayName最好还是加上。
* 静态方法要复制  
  无论PP还是II的方式，WrappedComponent的静态方法都不会复制，如果要用需要我们单独复制。
* refs不会传递。 意思就是HOC里指定的ref，并不会传递到子组件，如果你要使用最好写回调函数通过props传下去。
* 不要在render方法内部使用高阶组件。简单来说react的差分算法会去比较 NowElement === OldElement, 来决定要不要替换这个elementTree。也就是如果你每次返回的结果都不是一个引用，react以为发生了变化，去更替这个组件会导致之前组件的状态丢失。

## 解法对比及优缺点

**综上所述**：第一种只是关键字，而第二种说出其中的真正的原理

## 项目中体现经验的点

高阶组件的解耦和灵活性

## 论坛参考

# React与Vue的区别与联系原理：陈蔓杰项目经理

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 95% | 3星 | React与vue | 区别与联系 | React与Vue的区别与联系 |

## 通常解法

Vue是用于构建用户界面的渐进式框架，React是构建用户界面的组件化开发

## 通用大牛级解法

在渲染用户界面的时候，DOM的操作是最昂贵，不幸的是没有库可以让这些原始操作变得更快。我们能做的最好的就是：尽量减少DOM操作。Vue 和 React 都使用虚拟DOM来实现，并且两者工作一样好；

相同点：

1. .都支持服务器端渲染
2. 都使用虚拟DOM来实现
3. 都有Virtual DOM,组件化开发,通过props参数进行父子组件数据的传递,都实现webComponent规范
4. 只有框架的骨架，其他的功能如路由、状态管理等是框架分离的组件。
5. 都是JavaScript的UI框架，数据驱动视图，专注于创造前端的富应用
6. 都有支持native的方案,React的React native,Vue的weex
7. 都有管理状态，React有redux,Vue有自己的Vuex（自适应vue，量身定做）

在 React 中，所有的组件的渲染功能使用的都是JSX。JSX 是使用 XML 语法编写 Javascript 的一种语法糖，它可以使用的完整的编程语言JavaScript来构建视图页面，工具对JSX的支持相比于现有可用的其他Vue模板还是比较先进的，在 Vue 中我们采用的 Web 技术并在其上面扩展，使用Template在写模板的过程中，样式风格已确定和更少地涉及业务逻辑,一个模板总是被声明的,模板中任何 HTML 都是有效的,相比之下，React功能没有 Vue 模板系统的丰富。需要从组件文件中分离出HTML代码。

在性能方面，当我们考虑重新渲染功能。当组件的状态发生变化时，React的机制会触发整个组件树的重新呈现，并且由于 React 有大量的检查机制，能让它提供许多有用的警告和错误提示信息，但可能需要使用额外的属性来避免不必要地重新渲染子组件。虽然Vue的重新渲染功能是开箱即用的，但Vue提供了优化的重新渲染，其中系统在渲染过程中跟踪依赖关系并相应地工作。

在React中应用中的状态是（React）关键的概念。也有一些配套框架被设计为管理一个大的state对象，如Redux。此外，state对象在React应用中是不可变的，意味着它不能被直接改变（这也许不一定正确）。在React中你需要使用setState()方法去更新状态；在Vue中，state对象并不是必须的，数据由data属性在Vue对象中进行管理，而在Vue中，则不需要使用如setState()之类的方法去改变它的状态，在Vue对象中，data参数就是应用中数据的保存者。

不同点：

1. React严格上只针对MVC的view层,Vue则是MVVM模式
2. virtual DOM不一样,vue会跟踪每一个组件的依赖关系,不需要重新渲染整个组件树.
3. 而对于React而言,每当应用的状态被改变时,全部组件都会重新渲染,所以react中会需要shouldComponentUpdate这个生命周期函数方法来进行控制
4. 组件写法不一样, React推荐的做法是 JSX + inline style, 也就是把HTML和CSS全都写进JavaScript了,即'all in js';
5. Vue推荐的做法是webpack+vue-loader的单文件组件格式,即html,css,jd写在同一个文件;
6. 数据绑定: vue实现了数据的双向绑定,react数据流动是单向的
7. state对象在react应用中不可变的,需要使用setState方法更新状态;
8. 在vue中,state对象不是必须的,数据由data属性在vue对象中管理

## 解法对比及优缺点

**在以下场景中，Vue比反应更好：**

最新文档和更简单的语法。更小，更快，更灵活。丰富的HTML模板，易于开发。

**React比Vue.js好：**

需要构建移动应用程序。专业和出色的社区支持，以解决任何问题。需要构建大型应用程序。轻量级，易于版本迁移

## 项目中体现经验的点

Reat与Vue只有框架的骨架，其他的功能如路由、状态管理等是框架分离的组件。

## 论坛参考

# 解释一下跨域有几种方法？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 95% | 3星 | Js底层原理 | javaScript | 继承模式 |

## 通常解法

javascript中继承跟java中的继承不太一样，一般通过call()和apply()两种方式完成，js中的继承是以复制的形式完成的，复制一个父对象，而不像java中直接继承父对象，还有通过原型的方式完成继承，也有弊端，总之js中的继承只是形式上的对面向对象语言的一种模仿，本质上不是继承，但用起来效果是一样的。  
至于为什么要继承：通常在一般的项目里不需要，因为应用简单，但你要用纯js做一些复杂的工具或框架系统就要用到了，比如webgis、或者js框架如jquery、ext什么的，不然一个几千行代码的框架不用继承得写几万行，甚至还无法维护。

## 通用大牛级解法

在通用解法的基础上，我们实现继承的三种方式。

**原型继承**



这种原型继承的特点：既继承了父类的模板，又继承了父类的原型对象。优点是继承了父类的模板，又继承了父类的原型对象，缺点就是父类实例传参，不是子类实例化传参，不符合常规语言的写法。

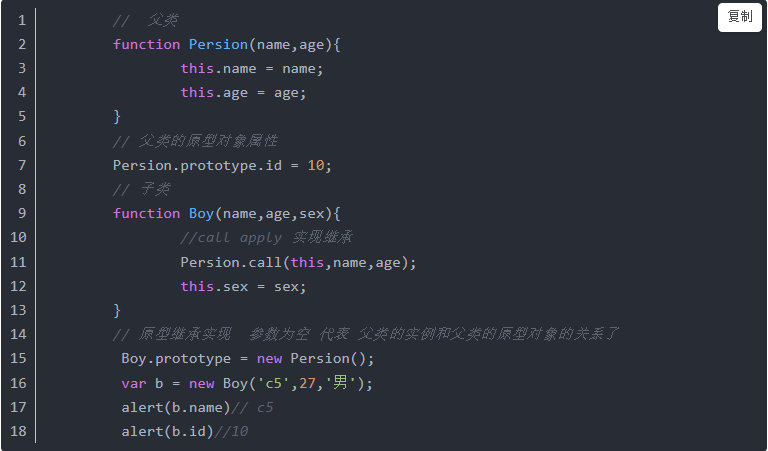
**类继承（借用构造函数的方式继承）**



这种原型继承的特点：继承了父类的模板，不继承了父类的原型对象。优点是方便了子类实例传参，缺点就是不继承了父类的原型对象

因而就可以将其占用的内存空间回收回来。这样，当垃圾收集器下次再运行时，它就会释放那些引用次数为零的值所占用的内存。

**混合继承（原型继承和类继承）**



这种原型继承的特点：既继承了父类的模板，又继承了父类的原型对象。优点方便了子类实例传参，缺点就是Boy.pertotype = new Persion()  函数又实例一次，函数内部变量又重复实例一次，大程序时候会很好性能。

## 解法对比及优缺点

对比以上解法，我们发现在通用解法中我们只说到JavaScript的继承本质和为什么要使用继承。而在大牛级解法中我们举出了三个常用继承的模式并加以代码说明，这样一来回答的更加具体和形象

## 延伸及扩展问题回答参考

上面是ES5的继承机制，对应即将广泛应用的ES6,我们也要了解它的新继承机制

Class之间可以通过extends关键字实现继承，这比ES5的通过修改原型链实现继承，要清晰和方便很多。

class ColorPoint extends Point {}

上面代码定义了一个ColorPoint类， 该类通过extends关键字， 继承了Point类的所有属性和方法。 但是由于没有部署任何代码， 所以这两个类完全一样， 等于复制了一个Point类。 下面， 我们在ColorPoint内部加上代码。

class ColorPoint extends Point {

constructor(x, y, color) {

super(x, y); // 调用父类的 constructor(x, y)

this.color = color;

}

toString() {

return this.color + ' ' + super.toString(); // 调用父类的 toString()

}

}

上面代码中， constructor方法和toString方法之中， 都出现了super关键字， 它在这里表示父类的构造函数， 用来新建父类的this对象。  
子类必须在constructor方法中调用super方法， 否则新建实例时会报错。 这是因为子类没有自己的this对象， 而是继承父类的this对象， 然后对其进行加工。 如果不调用super方法， 子类就得不到this对象。

class Point { /\* ... \*/ }

class ColorPoint extends Point {

constructor() {}

}

let cp = new ColorPoint(); // ReferenceError

上面代码中， ColorPoint继承了父类Point， 但是它的构造函数没有调用super方法， 导致新建实例时报错。  
ES5 的继承， 实质是先创造子类的实例对象this， 然后再将父类的方法添加到this上面（ Parent.apply(this)）。 ES6 的继承机制完全不同， 实质是先创造父类的实例对象this（ 所以必须先调用super方法）， 然后再用子类的构造函数修改this。  
如果子类没有定义constructor方法， 这个方法会被默认添加， 代码如下。 也就是说， 不管有没有显式定义， 任何一个子类都有constructor方法。

constructor(...args) {

super(...args);

}

另一个需要注意的地方是， 在子类的构造函数中， 只有调用super之后， 才可以使用this关键字， 否则会报错。 这是因为子类实例的构建， 是基于对父类实例加工， 只有super方法才能返回父类实例。

class Point {

constructor(x, y) {

this.x = x;

this.y = y;

}

}

class ColorPoint extends Point {

constructor(x, y, color) {

this.color = color; // ReferenceError

super(x, y);

this.color = color; // 正确

}

}

上面代码中， 子类的constructor方法没有调用super之前， 就使用this关键字， 结果报错， 而放在super方法之后就是正确的。  
下面是生成子类实例的代码。

let cp = new ColorPoint(25, 8, 'green');

cp instanceof ColorPoint // true

cp instanceof Point // true

上面代码中， 实例对象cp同时是ColorPoint和Point两个类的实例， 这与 ES5 的行为完全一致。

## 项目中体现经验的点

深刻理解JavaScript的继承机制，利用继承机制来优化和精简外面的代码；

## 论坛参考

<https://blog.csdn.net/caijixin/article/details/78295676>

<https://blog.csdn.net/qq_30100043/article/details/53542531>

# 解释一下jsonp的原理？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 90% | 3星 | 框架 | Vue | 数据管理 |

## 通常解法

vue的每个组件中都会维护一个自身数据管理属性：data和一个父组件传过来的数据属性：props，我们只要合理利用data和props就可以解决大部分应用场景，比如父子组件间通信和兄弟组件间通信。再复杂一点的场景我们可以用到Bus和本地存储技术，实现跨组件间的数据通信。

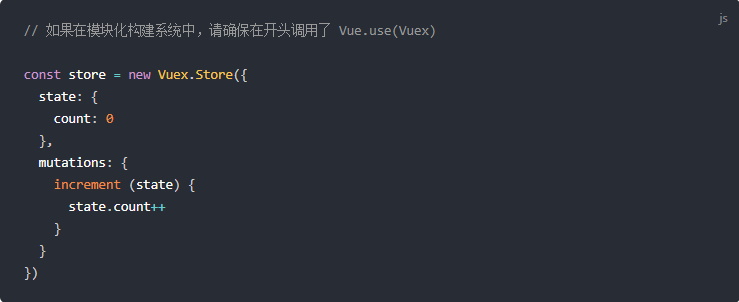
## 通用大牛级解法

在通用解法的基础上，介绍数据管理工具：Vuex

Vuex是一个专为 Vue.js 应用程序开发的状态管理模式。它采用集中式存储管理应用的所有组件的状态，并以相应的规则保证状态以一种可预测的方式发生变化。

每一个 Vuex 应用的核心就是 store（仓库）。“store”基本上就是一个容器，它包含着你的应用中大部分的状态 (state)。Vuex 和单纯的全局对象有以下两点不同：

* Vuex 的状态存储是响应式的。当 Vue 组件从 store 中读取状态的时候，若 store 中的状态发生变化，那么相应的组件也会相应地得到高效更新。
* 你不能直接改变 store 中的状态。改变 store 中的状态的唯一途径就是显式地提交 (commit) mutation。这样使得我们可以方便地跟踪每一个状态的变化，从而让我们能够实现一些工具帮助我们更好地了解我们的应用。



现在，你可以通过 store.state 来获取状态对象，以及通过 store.commit 方法触发状态变更：



我们通过提交 mutation 的方式，而非直接改变 store.state.count，是因为我们想要更明确地追踪到状态的变化。这个简单的约定能够让你的意图更加明显，这样你在阅读代码的时候能更容易地解读应用内部的状态改变。此外，这样也让我们有机会去实现一些能记录每次状态改变，保存状态快照的调试工具。有了它，我们甚至可以实现如时间穿梭般的调试体验。

由于 store 中的状态是响应式的，在组件中调用 store 中的状态简单到仅需要在计算属性中返回即可。触发变化也仅仅是在组件的 methods 中提交 mutation。

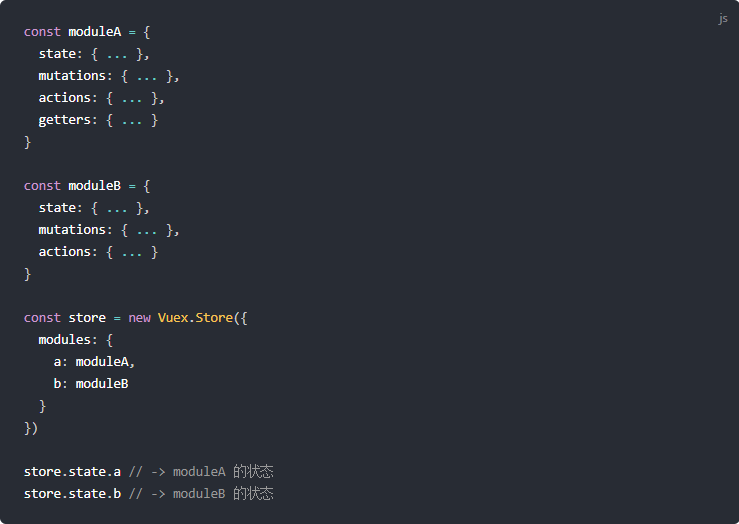
## 解法对比及优缺点

对比以上解法，我们发现在通用解法中我们只说到Vue自身的状态管理和BUS做组件间的通信，而在大牛级解法中我们给出了最新的类flux架构的vuex。Vuex既可以帮我们解决数据管理，并以相应的规则保证状态以一种可预测的方式发生变化来实现组件间的通信。这种解法让我们在回答问题时更有层次，我们的技术深度也能够更好的体现出来。

## 延伸及扩展问题回答参考

由于使用单一状态树，应用的所有状态会集中到一个比较大的对象。当应用变得非常复杂时，store 对象就有可能变得相当臃肿。

为了解决以上问题，Vuex 允许我们将 store 分割成模块（module）。每个模块拥有自己的 state、mutation、action、getter、甚至是嵌套子模块——从上至下进行同样方式的分割：



## 项目中体现经验的点

深刻理解vue的数据管理机制，在项目中管理好数据会帮助我们写出更加简洁高效的代码。

## 论坛参考

https://vuex.vuejs.org/zh/

<https://vuex.vuejs.org/zh/guide/modules.html>

[https://segmentfault.com/a/1190000010264128](https://vuex.vuejs.org/zh/guide/modules.html)

# Cookie的？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 95% | 3星 | Es6 | 异步 | Es6异步操作 |

## 通常解法

js是单线程运行的，分为异步任务和同步任务

## 通用大牛级解法

JavaScript引擎是单线程运行的,浏览器无论在什么时候都只且只有一个线程在运行JavaScript程序.浏览器的内核是多线程的，它们在内核制控下相互配合以保持同步，一个浏览器至少实现三个常驻线程：javascript引擎线程，GUI渲染线程，浏览器事件触发线程。这些异步线程都会产生不同的异步的事件.

1. javascript引擎是基于事件驱动单线程执行的，JS引擎一直等待着任务队列中任务的到来，然后加以处理，浏览器无论什么时候都只有一个JS线程在运行JS程序。
2. GUI渲染线程负责渲染浏览器界面，当界面需要重绘（Repaint）或由于某种操作引发回流(reflow)时,该线程就会执行。但需要注意 GUI渲染线程与JS引擎是互斥的，当JS引擎执行时GUI线程会被挂起，GUI更新会被保存在一个队列中等到JS引擎空闲时立即被执行。
3. 事件触发线程，当一个事件被触发时该线程会把事件添加到待处理队列的队尾，等待JS引擎的处理。这些事件可来自JavaScript引擎当前执行的代码块如setTimeOut、也可来自浏览器内核的其他线程如鼠标点击、AJAX异步请求等，但由于JS的单线程关系所有这些事件都得排队等待JS引擎处理。（当线程中没有执行任何同步代码的前提下才会执行异步代码）

当程序启动时, 一个进程被创建，同时也运行一个线程, 即为主线程，js的运行机制为单线程

程序中跑两个线程，一个负责程序本身的运行，作为主线程； 另一个负责主线程与其他线程的的通信，被称为“Event Loop 线程" 。每当遇到异步任务，交给 EventLoop 线程，然后自己往后运行，等到主线程运行完后，再去 EventLoop 线程拿结果。

1）所有任务都在主线程上执行，形成一个执行栈（execution context stack）。

2）主线程之外，还存在一个"任务队列"（task queue）。系统把异步任务放到"任务队列"之中，然后继续执行后续的任务。

3）一旦"执行栈"中的所有任务执行完毕，系统就会读取"任务队列"。如果这个时候，异步任务已经结束了等待状态，就会从"任务队列"进入执行栈，恢复执行。

4）主线程不断重复上面的第三步。

"回调函数"（callback），就是那些会被主线程挂起来的代码。异步任务必须指定回调函数，当异步任务从"任务队列"回到执行栈，回调函数就会执行。"任务队列"是一个先进先出的数据结构，排在前面的事件，优先返回主线程。主线程的读取过程基本上是自动的，只要执行栈一清空，"任务队列"上第一位的事件就自动返回主线程。

主线程从"任务队列"中读取事件，这个过程是循环不断的，所以整个的这种运行机制又称为**Event Loop。**

**从主线程的角度看，一个异步过程包括下面两个要素：**

发起函数(或叫注册函数)A

回调函数callbackFn

它们都是在主线程上调用的，其中注册函数用来发起异步过程，回调函数用来处理结果。

**异步进程有:**

类似onclick等，由浏览器内核的DOM binding模块处理，事件触发时，回调函数添加到任务队列中；

setTimeout等，由浏览器内核的Timer模块处理，时间到达时，回调函数添加到任务队列中；

Ajax，由浏览器内核的Network模块处理，网络请求返回后，添加到任务队列中。

例如setTimeout(fn, 1000)，其中的setTimeout就是异步过程的发起函数，fn是回调函数。用一句话概括：工作线程将消息放到消息队列，主线程通过事件循环过程去取消息。

消息队列：消息队列是一个先进先出的队列，它里面存放着各种消息。

事件循环：事件循环是指主线程重复从消息队列中取消息、执行的过程。

**流程如下:**

1. 主线程读取js代码, 形成相应的堆和执行栈, 执行同步任务
2. 当主线程遇到异步任务,，指定给异步进程处理, 同时继续执行同步任务
3. 当异步进程处理完毕后， 将相应的异步任务推入到任务队列首部
4. 主线程任务处理完毕后,，查询任务队列，则取出一个任务队列推入到主线程的执行栈

重复执行第2、3、4步，这就称为事件循环

## 解法对比及优缺点

**综上所述**：第一种只是关键字，而第二种说出其中的正真的执行原理

## 项目中体现经验的点

程序的执行顺序

## 论坛参考

# 说一下vue实现双向绑定原理？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 90% | 5星 | Javascript | 数据存储 | 存储 |

## 通常解法

恶意攻击者往Web页面里注入恶意Script代码，用户浏览这些网页时，就会执行其中的恶意代码，可对用户进行盗取cookie信息、会话劫持等各种攻击.；

## 通用大牛级解法

1. **1，XSS（Cross Site Scripting，跨站脚本攻击）**
2. 这是前端最常见的攻击方式，很多大型网站（如 Facebook）都被 XSS 攻击过。
3. 举一个例子，我在一个博客网站正常发表一篇文章，输入汉字、英文和图片，完全没有问题。但是如果我写的是恶意的 JS 脚本，例如获取到document.cookie然后传输到自己的服务器上，那我这篇博客的每一次浏览都会执行这个脚本，都会把访客 cookie 中的信息偷偷传递到我的服务器上来。
4. 其实原理上就是黑客通过某种方式（发布文章、发布评论等）将一段特定的 JS 代码隐蔽地输入进去。然后别人再看这篇文章或者评论时，之前注入的这段 JS 代码就执行了。JS 代码一旦执行，那可就不受控制了，因为它跟网页原有的 JS 有同样的权限，例如可以获取 server 端数据、可以获取 cookie 等。于是，攻击就这样发生了。
5. XSS的危害
6. XSS 的危害相当大，如果页面可以随意执行别人不安全的 JS 代码，轻则会让页面错乱、功能缺失，重则会造成用户的信息泄露。
7. 比如早些年社交网站经常爆出 XSS 蠕虫，通过发布的文章内插入 JS，用户访问了感染不安全 JS 注入的文章，会自动重新发布新的文章，这样的文章会通过推荐系统进入到每个用户的文章列表面前，很快就会造成大规模的感染。
8. 还有利用获取 cookie 的方式，将 cookie 传入入侵者的服务器上，入侵者就可以模拟 cookie 登录网站，对用户的信息进行篡改。
9. XSS的预防
10. 那么如何预防 XSS 攻击呢？—— 最根本的方式，就是对用户输入的内容进行验证和替换，需要替换的字符有：
11. & 替换为：&amp;
12. < 替换为：&lt;
13. > 替换为：&gt;
14. ” 替换为：&quot;
15. ‘ 替换为：&#x27;
16. / 替换为：&#x2f;
17. 替换了这些字符之后，黑客输入的攻击代码就会失效，XSS 攻击将不会轻易发生。
18. 除此之外，还可以通过对 cookie 进行较强的控制，比如对敏感的 cookie 增加http-only限制，让 JS 获取不到 cookie 的内容。
19. **2，CSRF（Cross-site request forgery，跨站请求伪造）**
20. CSRF 是借用了当前操作者的权限来偷偷地完成某个操作，而不是拿到用户的信息。
21. 例如，一个支付类网站，给他人转账的接口是<http://buy.com/pay?touid=999&money=100>，而这个接口在使用时没有任何密码或者 token 的验证，只要打开访问就直接给他人转账。一个用户已经登录了[http://buy.com](http://buy.com/)，在选择商品时，突然收到一封邮件，而这封邮件正文有这么一行代码<img src="<http://buy.com/pay?touid=999&money=100>"/>，他访问了邮件之后，其实就已经完成了购买。
22. CSRF 的发生其实是借助了一个 cookie 的特性。我们知道，登录了[http://buy.com](http://buy.com/)之后，cookie 就会有登录过的标记了，此时请求<http://buy.com/pay?touid=999&money=100>是会带着 cookie 的，因此 server 端就知道已经登录了。而如果在[http://buy.com](http://buy.com/)去请求其他域名的 API 例如<http://abc.com/api>时，是不会带 cookie 的，这是浏览器的同源策略的限制。但是 —— 此时在其他域名的页面中，请求<http://buy.com/pay?touid=999&money=100>，会带着[buy.com](http://buy.com/)的 cookie ，这是发生 CSRF 攻击的理论基础。
23. 预防 CSRF 就是加入各个层级的权限验证，例如现在的购物网站，只要涉及现金交易，肯定要输入密码或者指纹才行。除此之外，敏感的接口使用POST请求而不是GET也是很重要的。

## 3.解法对比及优缺点

通常解法，前端常见的安全问题。通用级解法，说出了如何模拟攻击，如何防御攻击

。

## 4.项目中体现经验的点

登录权限控制，富文本内容发布

## 5.论坛参考

# 解释一下 XSS与CSRF区别？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 50% | 3星 | 小程序 | 功能抽象 | 组件 |

## 通常解法

Promise 是异步编程的一种解决方案，比传统的解决方案——回调函数和事件——更合理和更强大。它由社区最早提出和实现，ES6 将其写进了语言标准，统一了用法，原生提供了Promise对象。

所谓Promise，简单说就是一个容器，里面保存着某个未来才会结束的事件（通常是一个异步操作）的结果。从语法上说，Promise 是一个对象，从它可以获取异步操作的消息。Promise 提供统一的 API，各种异步操作都可以用同样的方法进行处理。

基本用法：

const promise = new Promise(function(resolve, reject) {

// ... some code

if (/\* 异步操作成功 \*/){

resolve(value);

} else {

reject(error);

}

});

提供的方法

Promise.prototype.then()

Promise.prototype.catch()

Promise.prototype.finally()

Promise.all()

Promise.race()

## 通用大牛级解法

Promise对象的特点：

1. 对象的状态不受外界影响。Promise对象代表一个异步操作，有三种状态：pending（进行中）、fulfilled（已成功）和rejected（已失败）。
2. 一旦状态改变，就不会再变，任何时候都可以得到这个结果。Promise对象的状态改变，只有两种可能：从pending变为fulfilled和从pending变为rejected。只要这两种情况发生，状态就凝固了，不会再变了，会一直保持这个结果，这时就称为 resolved（已定型）。

有了Promise对象，就可以将异步操作以同步操作的流程表达出来，避免了层层嵌套的回调函数。此外，Promise对象提供统一的接口，使得控制异步操作更加容易。

Promise也有一些缺点。首先，无法取消Promise，一旦新建它就会立即执行，无法中途取消。其次，如果不设置回调函数，Promise内部抛出的错误，不会反应到外部。第三，当处于pending状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）。

如果某些事件不断地反复发生，一般来说，使用 Stream 模式是比部署Promise更好的选择。

除了promise也可以考虑es6的generator和async函数

## 解法对比及优缺点

* 普通解法只是promise对象的概念，以及基本的使用。
* 大牛级解法对promise的特点进行了解释，也总结了promise的缺点，并提供了更多的选项，体现了经验。

## 延伸及扩展问题回答参考

JavaScript的异步编程总结；

Ajax的异步原理是什么

## 项目中体现经验的点

深刻理解promise的特点和使用方法；

## 论坛参考

https://developers.weixin.qq.com/miniprogram/dev/framework/custom-component/events.html

# 解释一下函数节流？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 75% | 3星 | 虚拟DOM | 开发效率 | js |

## 通常解法

JavaScript eXtension，或更常见的JSX，是一个React扩展，允许我们编写看起来像 HTML的JavaScript 。

可以在 JSX 中使用 JavaScript 表达式。表达式写在花括号 {} 中；

在 JSX 中不能使用 if else 语句，但可以使用 conditional (三元运算) 表达式来替代；

React 推荐使用内联样式。我们可以使用 camelCase 语法来设置内联样式. React 会在指定元素数字后自动添加 px；

JSX 允许在模板中插入数组，数组会自动展开所有成员；

JSX事件绑定使用驼峰写法，比如<div onClick={()=>{}}></div>

## 通用大牛级解法

虽然JSX看起来像HTML，但它实际上只是React.createElement()的语法糖。当组件渲染时，它输出一个React元素树或该组件输出的HTML元素的虚拟表示。React然后将基于此React元素表示来确定对实际DOM所做的更改。

例如：

class Header extends React.Component {

render() {

return (

<h1>Hello World</h1>

);

}

}

其实对应于

class HelloWorld extends React.Component {

render() {

return (

React.createElement(

'h1',

{className: 'large'},

'Hello World'

)

);

}

}

就是说JSX最终会被babel等编译工具转换为普通的JS来让浏览器识别，而转换的结果是React.createElement()函数，该函数不会生成真实的DOM而是创建一个虚拟DOM，虚拟DOM本身就是一个对象。ReactDOM.render()将虚拟DOM渲染到视图上，会经过diff运算以减小不必要的DOM操作，从而提高页面性能。

## 解法对比及优缺点

* 普通解法只是概括jsx表面特点，以及如何使用。
* 大牛级解法对jsx的实质作了进一步阐述，充分体现开发人员对jsx语法的理解

## 延伸及扩展问题回答参考

虚拟DOM的概念被react框架发明，得到了社区的积极响应，目前主流应用框架都引入了虚拟DOM的概念，比如VUE也通过虚拟DOM+diff运算来提升页面性能

## 项目中体现经验的点

深刻理解JSX语法的特点和原理；

## 论坛参考

http://www.runoob.com/react/react-jsx.html

https://blog.csdn.net/tianzhw/article/details/78812475

<https://www.cnblogs.com/zourong/p/6043914.html>

# 解释事件“捕获”，“冒泡”执行原理？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 85% | 4星 | 数据管理 | 跨组建通信 | [reducers](http://www.redux.org.cn/docs/Glossary.html" \l "reducer) |

Redux 可以用这三个基本原则来描述：

## 单一数据源

**整个应用的 [state](http://www.redux.org.cn/docs/Glossary.html" \l "state) 被储存在一棵 object tree 中，并且这个 object tree 只存在于唯一一个 [store](http://www.redux.org.cn/docs/Glossary.html" \l "store) 中。**

这让同构应用开发变得非常容易。来自服务端的 state 可以在无需编写更多代码的情况下被序列化并注入到客户端中。由于是单一的 state tree ，调试也变得非常容易。在开发中，你可以把应用的 state 保存在本地，从而加快开发速度。此外，受益于单一的 state tree ，以前难以实现的如“撤销/重做”这类功能也变得轻而易举。

console.log(store.getState())

/\* Prints

{

visibilityFilter: 'SHOW\_ALL',

todos: [

{

text: 'Consider using Redux',

completed: true,

},

{

text: 'Keep all state in a single tree',

completed: false

}

]

}

\*/

## State 是只读的

**惟一改变 state 的方法就是触发 [action](http://www.redux.org.cn/docs/Glossary.html" \l "action)，action 是一个用于描述已发生事件的普通对象。**

这样确保了视图和网络请求都不能直接修改 state，相反它们只能表达想要修改的意图。因为所有的修改都被集中化处理，且严格按照一个接一个的顺序执行，因此不用担心 race condition 的出现。 Action 就是普通对象而已，因此它们可以被日志打印、序列化、储存、后期调试或测试时回放出来。

store.dispatch({

type: 'COMPLETE\_TODO',

index: 1

});

store.dispatch({

type: 'SET\_VISIBILITY\_FILTER',

filter: 'SHOW\_COMPLETED'

});

## 使用纯函数来执行修改

**为了描述 action 如何改变 state tree ，你需要编写 [reducers](http://www.redux.org.cn/docs/Glossary.html" \l "reducer)。**

Reducer 只是一些纯函数，它接收先前的 state 和 action，并返回新的 state。刚开始你可以只有一个 reducer，随着应用变大，你可以把它拆成多个小的 reducers，分别独立地操作 state tree 的不同部分，因为 reducer 只是函数，你可以控制它们被调用的顺序，传入附加数据，甚至编写可复用的 reducer 来处理一些通用任务，如分页器。

function visibilityFilter(state = 'SHOW\_ALL', action) {

switch (action.type) {

case 'SET\_VISIBILITY\_FILTER':

return action.filter

default:

return state

}

}

function todos(state = [], action) {

switch (action.type) {

case 'ADD\_TODO':

return [

...state,

{

text: action.text,

completed: false

}

]

case 'COMPLETE\_TODO':

return state.map((todo, index) => {

if (index === action.index) {

return Object.assign({}, todo, {

completed: true

})

}

return todo

})

default:

return state

}

}

import { combineReducers, createStore } from 'redux'

let reducer = combineReducers({ visibilityFilter, todos })

let store = createStore(reducer)

## **延伸及扩展问题回答参考**

## **项目中体现经验的点**

## **论坛参考**

# 解释一下页面重构操作？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 40% | 5星 | redux中间件 | 异步流程管理了 | 数据容器 |

## 通常解法

Saga是一个可以用来处理复杂的异步逻辑的模块，并且由redux的action触发，Saga副作用就是在action触发reducer之后执行的一些动作， 这些动作包括但不限于，连接网络，io读写，触发其他action。并且，因为Saga的副作用是通过redux的action触发的，每一个action，saga都会像reducer一样接收到。并且通过触发不同的action, 我们可以控制这些副作用的状态， 例如，启动，停止，取消。通常配置Saga中间件由2部分组成：worker saga和watcher saga。watcher saga用于订阅action，worker saga用于根据action执行相应的操作。

所有的saga都是基于ES6的generator，generator函数可以将异步逻辑转为同步逻辑来编写。

例如：

watcher saga：function\* watch(){

yield takeEvery(‘action type’, worker saga)；

}

worker saga: function\* worker(){

yield call()

yield put()

}；

## 通用大牛级解法

Saga可以实现复杂异步流程管理，适合在大型项目且业务场景复杂的情况下使用。在业务逻辑层，可以简化代码，使代码更加容易阅读。 在重用方面，解耦显示层和业务层之后， 代码的重用度也得到了提升。使用基础API如take, put, call, fork, all等可以实现多流程控制，包括顺序执行，并行执行和竞争执行等。相比于redux-thunk中间件只是实现异步流程控制反转dispatch(function)来说，避免了难以测试的问题，逻辑流程也更加清晰。

Saga的应用简单总结来说就是：

视图=>dispatch(action)=>saga中间件:订阅action(watcher)+ worker + dispatch(action)=>数据容器redux(reducer)。

虽然saga可以起很大的效果，但是也有弊端：

* redux-saga模型的理解和学习需要投入很多精力
* 因为需要用action触发，所以会产生很多对于reducer无用的action, 但是reducer一样会跑一轮，虽然目前没有观测到性能下降，但还是有计算开销
* 在action的定义上要谨慎，避免action在saga和reducer之间重复触发，造成死循环
* 增加了项目目录结构复杂度和维护成本

。

## 解法对比及优缺点

普通解法只是对redux-saga基本使用的介绍，仅限于知道怎么用，不能说明redux-saga的适用场景，以及更复杂场景的使用情况。

大牛解法则清楚地说明了redux-saga的应用流程，心里明确什么时候需要使用saga。并且对潜在问题做了说明，同时对比了其他类似的中间件，体现了丰富的应用经验

## 延伸及扩展问题回答参考

对比说明redux-saga和redux-thunk的特点？

越是用来解决具体问题的技术，使用起来越容易，越高效，学习成本越低；越是用来解决宽泛问题的技术，使用起来越难，学习成本越高。 thunk解决的是很具体的一个问题，就是在action到达reducer之前做一些其他的业务，比如fetch后端, 它在做这件事的上很高效。而Saga解决的问题要更宽泛一些，因为saga只是拦截了action，至于做什么，开发者需要自己来考虑，可以是fetch后端，也可以是更新redux store, 甚至可以执行action带进来的callback。 很显然对于一个业务层来说,saga会是一个更合适的选择，但同时也带来了学习成本的提高。

## 项目中体现经验的点

深刻理解项目异步流程管理问题，以及选择正确的解决方案；

## 论坛参考

<https://redux-saga-in-chinese.js.org/docs/api/>

1. <https://www.jianshu.com/p/89ed2a01a3db>

# 谈谈对fetch对的看法？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 70% | 4星 | 前端 | 数据安全 | 数据安全 |

## 通常解法

Node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行环境。   
Node.js 使用了一个事件驱动、非阻塞式 I/O 的模型，使其轻量又高效。   
Node.js 的包管理器 npm，是全球最大的开源库生态系统。

Node解决了js跑在服务端的一个空缺

## 通用大牛级解法

加上通用解法后

可以解决高并发，它是单线程，当访问量很多时，将访问者分配到不同的内存中，不同的内存区做不同的事，以快速解决这个线程。就像医院的分科室看病人。效率快，但消耗内存大、异步和事件驱动。概扩起来就三点：单线程、异步I/O、事件驱动。nodejs离不开ChormeV8引擎，也就是V8引擎是来解释javascript。用nodejs来搭建高性能的Web服务器，因此node.js是基于服务器端的javascript

node主要应用场景是在大前端，阿里的思路是比较合适的，但是必须要注意，绝对不能让node做太多的业务逻辑，他只适合接受人家生成好的数据，然后或渲染后，或直接发送到客户端。如果让node做复杂的业务逻辑，那会得不偿失的

回调模式下的异步是有明显缺陷的，程序的执行顺序必须依靠回调来保证，没有层层回调，就没有可以保障的逻辑顺序，这也就注定了，node不能做复杂的业务逻辑。javascript语言本身也一直在和回调做斗争，promise，generator都可以将回调包装起来，在代码的某个部分形成形式同步，但是这种模式进化的还不完全，还不能做到与回调完全割裂，做到完全的形式同步。但是形式同步肯定是发展的方向，这种模式即可以获得异步的好处，又可以有效回避回调带来的编程困难，在业务逻辑上可以更简单的表达。

就现在的环境来说，大家的思路还没转过弯，对回调的批评认为都是不好的，这些人是不敢面对现实，javascript都在变，这些人的脑子却不肯变，还以为回调就代表异步。

Node.js的发展给javascript注入了新的生命力

至于javascript的垢病，个人感觉，不在它的callback而在它的随意性，随意到想怎么写都行，但正是这点给它带来了惊人的开发效率，做好代码规范和文档工作可以减少javascript的随意性带来的负面影响。

WEB端、移动端、桌面端、甚至嵌入式，javascript已经无处不在。接下来，ES6的实现会让众多习惯同步或者不喜欢回调的开发人员能够更快地上手javascript写出符合他们思维习惯的代码，这些开发人员会是更大的群体，那么也许javascript会横扫应用开发也不一定。

所以，javascript很有前途，那Node.js自然就有前途。

## 解法对比及优缺点

大牛级解法：

优点：可以比较深入的解释清楚node底层原理和带来的一些良性循环；

缺点：如果对基础知识不牢固，可能会带来一系列的问题；

通用解法：

优点：可以比较概括的回答面试的问题

缺点：不够相信，不够全面

## 延伸及扩展问题回答参考

**问题**：目前前端对node的应用场景有哪些？

解答： 前端自动化，客户端服务都在大规模的使用node进行问题的解决！

## 。项目中体现经验的点

**可以解决前端本地开发的服务问题，可以实现接口的代理，可以实现客户端服务等**

## 论坛参考

<https://blog.csdn.net/hsj1669666567/article/details/79568545>

https://cnodejs.org/topic/53c88a83c9507b4044b0a78a

# 解释一下同源策略？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 95% | 3星 | Es6 | 异步 | Es6异步操作 |

## 通常解法

vue生命周期一共有11个，

**beforeCreate,created**

**beforeMount,mounted**

**beforeUpdate, updated**

**actived** , deactivated

beforeDestroy, destroyed

errorCaptured

## 通用大牛级解法

vue的生命周期主要分为几个简单，数据初始化，dom挂载，数据更新，组件卸载，在一个就是开启了组件缓存的时候，会有组件启用和组件停用阶段，每个阶段都去前后两个钩子除了缓存的那俩

数据初始化阶段

beforeCreate：在实例初始化之后，数据观测 (data observer) 和 event/watcher 事件配置之前被调用。

created：实例已经创建完成之后被调用。在这一步，实例已完成以下的配置：数据观测 (data observer)，属性和方法的运算，watch/event 事件回调。然而，挂载阶段还没开始，$el 属性目前不可见。

dom挂载阶段

beforeMount：在挂载开始之前被调用：相关的 render 函数首次被调用。

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。

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 替换掉 mounted

数据跟新阶段

beforeUpdate：数据更新时调用，发生在虚拟 DOM 重新渲染和打补丁之前。

updated：由于数据更改导致的虚拟 DOM 重新渲染和打补丁，在这之后会调用该钩子。

当这个钩子被调用时，组件 DOM 已经更新，所以你现在可以执行依赖于 DOM 的操作。然而在大多数情况下，你应该避免在此期间更改状态。如果要相应状态改变，通常最好使用计算属性或 watcher 取而代之。

updated 不会承诺所有的子组件也都一起被重绘。如果你希望等到整个视图都重绘完毕，可以用 vm.$nextTick 替换掉 updated：

缓存启用的时候会有下面两个钩子

activated：keep-alive 组件激活时调用。

deactivated：keep-alive 组件停用时调用。

组件卸载的时候：

beforeDestroy：实例销毁之前调用。在这一步，实例仍然完全可用。

destroyed：Vue 实例销毁后调用。调用后，Vue 实例指示的所有东西都会解绑定，所有的事件监听器会被移除，所有的子实例也会被销毁。

## 解法对比及优缺点

**综上所述**：第一种只是说出了生命周期方法，而第二种说出生命周期的执行作用

## 延伸及扩展问题回答参考

**问题**： 数据请求应该在那个生命周期，

**解答**：在created里面

## 项目中体现经验的点

组件中必须的内容

## 论坛参考

<https://cn.vuejs.org/v2/api>

# 说一下jsonp工作原理？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 90% | 5星 | Javascript | 数据存储 | 存储 |

## 通常解法

Vuex 是适用于 Vue.js 应用的状态管理库，为应用中的所有组件提供集中式的状态存储与操作，保证了所有状态以可预测的方式进行修改；

state: state 定义了应用状态的数据结构，同样可以在这里设置默认的初始状态。

actions:Actions 即是定义提交触发更改信息的描述，常见的例子有从服务端获取数据，在数据获取完成后会调用store.commit()来调用更改 Store 中的状态。可以在组件中使用dispatch来发出 Actions。

mutations: 调用 mutations 是唯一允许更新应用状态的地方。

getters: Getters 允许组件从 Store 中获取数据，譬如我们可以从 Store 中的 projectList 中筛选出已完成的项目列表

modules: modules 对象允许将单一的 Store 拆分为多个 Store 的同时保存在单一的状态树中。随着应用复杂度的增加，这种拆分能够更好地组织代码

但是vuex也有缺点就是，vuex中保存的数据是和网页的生命周期同步的，当执行页面刷新的时候vuex中所有数据都会消失复位到初始状态，所以不太适合做有分享页面的数据交互（在这种项目中vuex只适合数据的集中管理，不适合数据的存储，这种情况一般是使用路由传递参数会好一些），适合后台管理系统多一些，后台管理系统一般都是公司内部使用；

## 通用大牛级解法

Vuex 是专门针对vue开发的应用状态管理库，为应用中的所有组件提供集中式的状态存储与操作，保证了所有状态以可预测的方式进行修改；

state: state 定义了应用状态的数据结构，同样可以在这里设置默认的初始状态。

actions:Actions 即是定义提交触发更改信息的描述，常见的例子有从服务端获取数据，在数据获取完成后会调用store.commit()来调用更改 Store 中的状态。可以在组件中使用dispatch来发出 Actions。

mutations: 调用 mutations 是唯一允许更新应用状态的地方。

getters: Getters 允许组件从 Store 中获取数据，譬如我们可以从 Store 中的 projectList 中筛选出已完成的项目列表

modules: modules 对象允许将单一的 Store 拆分为多个 Store 的同时保存在单一的状态树中。随着应用复杂度的增加，这种拆分能够更好地组织代码

在项目vue中主要使用vuex需要注意的是vuex相当于一个全局的状态，所以不要在vuex保存大量的数据，只需要保存全局通用数据就可以了，还有就是当刷新页面的时候由于是js对象所以对象中的数据都会回归初始化，所以一般都需要配合本地存储一块使用，在就是vuex针对于vue的数据检测做了适配，所以在vue中使用vuex做状态管理还是比较不错的

## 解法对比及优缺点

通常解法，vuex的规则解释。通用级解法，说出了vuex的特性，优缺点，以及

。

## 项目中体现经验的点

登录判断，本地数据线上数据的同步

## 论坛参考

<https://vuex.vuejs.org/>

# JavaScript作用域链？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 50% | 3星 | 小程序 | 功能抽象 | 组件 |

## 通常解法

从小程序基础库版本 1.6.3 开始，小程序支持简洁的组件化编程。所有自定义组件相关特性都需要基础库版本 1.6.3 或更高。

开发者可以将页面内的功能模块抽象成自定义组件，以便在不同的页面中重复使用；也可以将复杂的页面拆分成多个低耦合的模块，有助于代码维护。自定义组件在使用时与基础组件非常相似。

在自定义组件的 js 文件中，需要使用 Component() 来注册组件，并提供组件的属性定义、内部数据和自定义方法。

组件的属性值和内部数据将被用于组件 wxml 的渲染，其中，属性值是可由组件外部传入的

组件构造器包含的属性和方法：

properties，data，methods，behaviors（mixin），created，attached，ready，moved，detached，relations等等

## 通用大牛级解法

理解小程序组件可以对比vue组件或者react组件，页面中多组件嵌套肯定会涉及跨组件传值，那么小程序的组件传值是如何实现的。Vue或者react都属于单向数据流的数据管理方式，小程序组件也是类似的。通过在组件上添加自定义属性，属性值可以是父组件的任意变量，在组建内使用properties属性来接收。例如：<test data=”{{count}}” bindtest=”func”></test>

Component({

Properties:{

data:{

Type:String,

Value:’’,

Observer:function(){}

}

}

})

这是父子组件传值，其中type限制组件接受值的格式，value可以设置该属性对应的默认值，observer类似于vue的watch，可以订阅属性值的变化，进而做相应的操作。

小程序组件由子组件向父组件传值则用的的事件派发的方式通过triggerEvent方法触发，并在组件标签上bind事件监听，执行回调函数，来实现子父传值。类似于vue的$emit, v-on组合

除了这些传值方式以外，还有全局对象挂载的方式。任何页面都可以通过getApp()得到app实例，该app实例是小程序的全局对象，拥有上帝视角，并且可以添加自定义属性，利用这点也能实现跨组件传值，不再限于父子组件之间。

## 解法对比及优缺点

* 普通解法只是小程序组件的概念和api，以及基本的使用。
* 大牛级解法对小程序组件的数据流进行了梳理，解释清楚了实际操作中会遇到的具体问题，以及解决方式，体现了丰富的经验，以及数据流概念

## 延伸及扩展问题回答参考

对比vue，react，小程序组件各自的特点，总结它们共性和不同点

## 项目中体现经验的点

深刻理解小程序组件的特点和使用方法；

## 论坛参考

## <https://developers.weixin.qq.com/miniprogram/dev/framework/custom-component/events.html>

# 谈谈RESTful？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 70% | 3星 | 全局状态管理 | 状态管理 | redux |

react-redux的作用是连接（connect）store和容器组件的。store是redux提供的，容器组件是react提供的。

## 组织应用的组件

* 组织应用的组件
  + 容器组件
  + 展示组件

容器组件：位于应用最顶层的组件，用来与redux连接的。从redux中获取数据作为props。

展示组件：位于应用的中间或者子组件，是纯粹的组件，与redux没有关系。他们从自己的父组件获取数据作为props，他们的共同根组件是应用的唯一的容器组件。展示组件可以维持少量的自身状态信息。

## 连接Store与组件

react-redux仅仅提供两个关键模块：Provider和connect。

源码：

import Provider from './components/Provider'

import connect from './components/connect'

export { Provider, connect }

* Provider：是一个组件，接受一个store属性和一个子组件（也就是上面说到的：store是redux提供的，容器组件是是react提供的。）

## 通常解法

ReactDOM.render(

<Provider store={store}>

<Handler routerState={routerState} />

</Provider>,

document.getElementById('root')

);

* connect([mapStateToProps], [mapDispatchToProps], [mergeProps], [options])：connect返回一个函数，它接受一个React组件的构造函数作为连接对象，最终返回连接好的组件构造函数。

import \* as actionCreators from './actionCreators'

function mapStateToProps(state) {

return { todos: state.todos }

}

export default connect(mapStateToProps, actionCreators)(MyRootComponent)

## 延伸及扩展问题回答参考

## 项目中体现经验的点

## 论坛参考

# 简述cookie？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 70% | 4星 | 性能优化 | javaScript | 性能优化 |

## 通常解法

**从网络加载方面来考虑**

* 减少HTTP请求次数

建议尽可能的根据需要去合并静态资源图片、JavaScript代码和CSS文件，减少页面请求数，这样可以缩短页面首次访问的等待时间，另外也要尽量的避免重复资源，防止增加多余的请求

* 减少HTTP请求大小

除了减少请求资源数，也要减少每个http请求的大小。比如减少没必要的图片，JS，CSS以及HTML等，对文件进行压缩优化，开启GZIP压缩传输内容，缩短网络传输等待延迟

* 将CSS和JS放到外部文件中，避免使用style和script标签引入

在HTML文件中引入外部的资源可以有效利用浏览器的静态资源缓存。有时候在移动端对请求数比较在意的会为了减少请求把CSS和JS文件直接写到HTML里边，具体根据CSS和JS文件大小和业务场景来分析。如果CSS和JS文件内容较多，逻辑比较复杂，建议放到外部引入

* 避免页面中空的href和src

当link标签的href属性为空，或者script、img、iframe标签的src属性为空的时候，浏览器在渲染的过程中还是会把href和src的空内容进行加载，直到加载失败。这样就阻塞了页面中其他资源的下载进程，并且最后加载的内容是无效的，因此要尽量避免。

* 为HTML指定Cache-Control或者Expires

为HTML指定Cache-Control或者Expires可以将HTML内容缓存起来，避免频繁向服务器发送请求。在页面Cache-Control或Expires头部又消失，浏览器会直接从缓存读取内容，不向服务器发送请求

* 静态资源不同域名存放

浏览器在同一时刻向同一个域名请求文件的并行下载数是有限的，因此可以理由多个域名的主机来存放不同的静态资源，增大页面加载时资源的并行下载数。

* 使用get请求

POST请求会首先发送文件头，然后发送HTTP正文的数据。而使用GET只发送头部，所以在拉取数据时使用GET请求效率更高

* 消除阻塞页面的CSS和JS

对于页面中加载时间过长的CSS或JS文件，需要进行合理的拆分或者延后加载，保证关键的资源能快速加载完成

* 按需加载

## 通用大牛级解法

不仅考虑到网络请求方面，同时也考虑到页面渲染方面

* 把CSS资源引用放到HTML文件顶部

这样浏览器可以优先下载CSS并尽快完成页面渲染

* JavaScript文件引用放到HTML文件底部

可以防止JavaScript的加载和解析执行对页面渲染造成阻塞。由于JavaScript资源默认是解析阻塞的，除非被标记为异步或者通过其他的方式异步加载，否则会阻塞HTML DOM解析和CSS渲染过程

* 不要在HTML中直接缩放图片

在HTML中直接缩放图片会导致页面内容的重排重绘，此时可能会使页面中的其他操作产生卡顿，因此要尽量减少在页面中直接进行图片缩放

* 减少DOM元素数量和深度

HTML中标签元素约的，标签的层级越深，浏览器解析DOM并绘制到浏览器中说花的时间就越长。

* 尽量避免使用table、iframe等慢元素

<table>内容的渲染是讲table的DOM渲染树全部生成完并一次性绘制到页面上，所以在长表格渲染时很耗性能，应该尽量避免使用，可以考虑用ul代替。尽量使用异步的方式动态的加载iframe，因为iframe内资源的下载进程会阻塞父页面静态资源的下载以及HTML DOM的解析

* 避免运行耗时的JavaScript

长时间运行的JavaScript会阻塞浏览器构建DOM树、DOM渲染树、渲染页面。所以任何与页面初次渲染无关的逻辑功能都应该延迟加载执行，这和JavaScript资源的异步加载思路一致

## 解法对比及优缺点

前端优化的策略有很多，主要包括网络加载，页面渲染，CSS优化，JS执行优化，缓存，图片，协议几大类；

网络请求是影响前端性能的一个主要因素，当然页面渲染也是一个非常重要的因素；因此我们应该针对这个两个方面都进行优化；

## 延伸及扩展问题回答参考

* 请减少HTTP请求基本原理：

在浏览器(客户端)和服务器发生通信时，就已经消耗了大量的时间，尤其是在网络情况比较糟糕的时候，这个问题尤其的突出。

一个正常HTTP请求的流程简述：如在浏览器中输入"www.xxxxxx.com"并按下回车，浏览器再与这个URL指向的服务器建立连接，然后浏览器才能向服务器发送请求信息，服务器在接受到请求的信息后再返回相应的信息，浏览器接收到来自服务器的应答信息后，对这些数据解释执行。

而当我们请求的网页文件中有很多图片、CSS、JS甚至音乐等信息时，将会频繁的与服务器建立连接，与释放连接，这必定会造成资源的浪费，且每个HTTP请求都会对服务器和浏览器产生性能负担。

网速相同的条件下，下载一个100KB的图片比下载两个50KB的图片要快。所以，请减少HTTP请求。

解决办法：

合并图片（css sprites），合并CSS和JS文件；图片较多的页面也可以使用 lazyLoad 等技术进行优化。

* 请正确理解 Repaint 和 Reflow

基本原理：

Repaint(重绘)就是在一个元素的外观被改变，但没有改变布局(宽高)的情况下发生，如改变visibility、outline、背景色等等。

Reflow(重排)就是DOM的变化影响到了元素的几何属性（宽和高），浏览器会重新计算元素的几何属性，会使渲染树中受到影响的部分失效，浏览器会验证DOM树上的所有其它结点的visibility属性，这也是Reflow低效的原因。如：改变窗囗大小、改变文字大小、内容的改变、浏览器窗口变化，style属性的改变等等。如果Reflow的过于频繁，CPU使用率就会噌噌的往上涨，所以前端也就有必要知道 Repaint 和 Reflow的知识。

减少性能影响的办法：

上面提到通过设置style属性改变结点样式的话，每设置一次都会导致一次reflow，所以最好通过设置class的方式； 有动画效果的元素，它的position属性应当设为fixed或absolute，这样不会影响其它元素的布局；如果功能需求上不能设置position为fixed或absolute，那么就权衡速度的平滑性。

总之，因为 Reflow 有时确实不可避免，所以只能尽可能限制Reflow的影响范围。

## 项目中体现经验的点

在操作页面的dom时，方式很多，但是我们应该从影响前端性能的这个方面着手去解决这个问题，避免过多的重排，操作dom元素不要过深

## 6.论坛参考

https://www.cnblogs.com/liulilin/p/7245125.html

<https://www.jianshu.com/p/ead7dab72cd6?mType=Group>

<https://blog.csdn.net/w2326ice/article/details/64122372>

# 谈谈vuejs变化检测问题？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 95% | 3星 | 前端 | 移动端适配 | 移动端适配 |

## 通常解法

通过百分比进行单位的换算；

通过媒介查询进行单位的设置；

使用媒介查询配合rem

@media screen and (min-width:360px) and (max-width:374px) and (orientation:portrait) {

html { font-size: 703%; }

}

@media screen and (min-width:375px) and (max-width:383px) and (orientation:portrait) {

html { font-size: 732.4%; }

}

@media screen and (min-width:384px) and (max-width:399px) and (orientation:portrait) {

html { font-size: 750%; }

}

@media screen and (min-width:400px) and (max-width:413px) and (orientation:portrait) {

html { font-size: 781.25%; }

}

@media screen and (min-width:414px) and (max-width:431px) and (orientation:portrait){

html { font-size: 808.6%; }

}

@media screen and (min-width:432px) and (max-width:479px) and (orientation:portrait){

html { font-size: 843.75%; }

}

## 通用大牛级解法

把上面的解法说一遍………

**网易手机端**

body的width都是7.5rem。很明显，目前网易的手机端设计稿是基于iPhone6，750（设计师给的设计稿是物理分辨率，会是我们写样式的逻辑分辨率的两倍，如果给的设计稿是640，那么是基于iPhone5，320），且基准值是100px（750/7.5=100）。这个基准值很关键，后面的css换算，都和这个基准值有关。动态font-size： 我们看到图1、图2、图3的font-size都有根据屏幕大小而动态改变，可以推算出公式：

屏幕宽度/设计稿rem宽度=页面动态font-size值（如：375/7.5=50）

获取到这个值，再赋给html元素的style：

document.documentElement.style.fontSize = document.documentElement.clientWidth / 7.5 + ‘px‘;

这样就设置好了每个页面的根fonts-size，因为rem单位是基于根font-size，因此只要确定一种设计稿对应手机的换算，其余屏幕尺寸均可自动适配。

上面我们得出设计稿换算rem的基准值是100，因此只需要把设计稿上的px值除以100即为我们要的rem值。

> Px/100=rem，所以100px=1rem,25px=0.25rem

**淘宝手机端**

引入： 直接引用阿里的CDN文件（或下载到本地引入）

<script src="http://g.tbcdn.cn/mtb/lib-flexible/0.3.4/??flexible\_css.js,flexible.js"></script>

设定： 页面不要设定 。Flexible会自动设定每个屏幕宽度的根font-size、动态viewport、针对Retina屏做的dpr。

换算： 假设拿到的设计稿和上述网易的一样都是750，Flexible会把设计稿分为10份，可以理解为页面width=10rem，即1rem=75px，所以根font-size（基准值）=75px。

## 解法对比及优缺点

**通用方案**

1、设置根font-size：625%（或其它自定的值，但换算规则1rem不能小于12px）

2、通过媒体查询分别设置每个屏幕的根font-size

3、css直接除以2再除以100即可换算为rem。

优：有一定适用性，换算也较为简单。

劣：有兼容性的坑，对不同手机适配不是非常精准；需要设置多个媒体查询来适应不同手机，单某款手机尺寸不在设置范围之内，会导致无法适配。

**网易方案**

1、拿到设计稿除以100，得到宽度rem值

2、通过给html的style设置font-size，把1里面得到的宽度rem值代入x  document.documentElement.style.fontSize = document.documentElement.clientWidth / x + ‘px‘;

3、设计稿px/100即可换算为rem

优：通过动态根font-size来做适配，基本无兼容性问题，适配较为精准，换算简便。

劣：无viewport缩放，且针对iPhone的Retina屏没有做适配，导致对一些手机的适配不是很到位

**手淘方案**

1、拿到设计稿除以10，得到font-size基准值

2、引入flexible

3、不要设置meta的viewport缩放值

4、设计稿px/ font-size基准值，即可换算为rem

优：通过动态根font-size、viewpor、dpr来做适配，无兼容性问题，适配精准。

劣：需要根据设计稿进行基准值换算，在不使用sublime text编辑器插件开发时，单位计算复杂。

## 延伸及扩展问题回答参考

**问题**：移动端项目对于背景图的加载

**解答**： 使用的网易解决方案可以实现雪碧图的使用；

项目中体现经验的点

使用网易的解决的方案可以使用的移动端适配更加精确

## 论坛参考

<https://www.cnblogs.com/liangxuru/p/6970629.html>

<https://www.cnblogs.com/cench/p/5314044.html>

# 谈谈对nodejs对的看法？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 70% | 4星 | 前端 | 数据安全 | 数据安全 |

## 通常解法

Node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行环境。   
Node.js 使用了一个事件驱动、非阻塞式 I/O 的模型，使其轻量又高效。   
Node.js 的包管理器 npm，是全球最大的开源库生态系统。

Node解决了js跑在服务端的一个空缺

## 通用大牛级解法

加上通用解法后

可以解决高并发，它是单线程，当访问量很多时，将访问者分配到不同的内存中，不同的内存区做不同的事，以快速解决这个线程。就像医院的分科室看病人。效率快，但消耗内存大、异步和事件驱动。概扩起来就三点：单线程、异步I/O、事件驱动。nodejs离不开ChormeV8引擎，也就是V8引擎是来解释javascript。用nodejs来搭建高性能的Web服务器，因此node.js是基于服务器端的javascript

node主要应用场景是在大前端，阿里的思路是比较合适的，但是必须要注意，绝对不能让node做太多的业务逻辑，他只适合接受人家生成好的数据，然后或渲染后，或直接发送到客户端。如果让node做复杂的业务逻辑，那会得不偿失的

回调模式下的异步是有明显缺陷的，程序的执行顺序必须依靠回调来保证，没有层层回调，就没有可以保障的逻辑顺序，这也就注定了，node不能做复杂的业务逻辑。javascript语言本身也一直在和回调做斗争，promise，generator都可以将回调包装起来，在代码的某个部分形成形式同步，但是这种模式进化的还不完全，还不能做到与回调完全割裂，做到完全的形式同步。但是形式同步肯定是发展的方向，这种模式即可以获得异步的好处，又可以有效回避回调带来的编程困难，在业务逻辑上可以更简单的表达。

就现在的环境来说，大家的思路还没转过弯，对回调的批评认为都是不好的，这些人是不敢面对现实，javascript都在变，这些人的脑子却不肯变，还以为回调就代表异步。

Node.js的发展给javascript注入了新的生命力

至于javascript的垢病，个人感觉，不在它的callback而在它的随意性，随意到想怎么写都行，但正是这点给它带来了惊人的开发效率，做好代码规范和文档工作可以减少javascript的随意性带来的负面影响。

WEB端、移动端、桌面端、甚至嵌入式，javascript已经无处不在。接下来，ES6的实现会让众多习惯同步或者不喜欢回调的开发人员能够更快地上手javascript写出符合他们思维习惯的代码，这些开发人员会是更大的群体，那么也许javascript会横扫应用开发也不一定。

所以，javascript很有前途，那Node.js自然就有前途。

## 解法对比及优缺点

大牛级解法：

优点：可以比较深入的解释清楚node底层原理和带来的一些良性循环；

缺点：如果对基础知识不牢固，可能会带来一系列的问题；

通用解法：

优点：可以比较概括的回答面试的问题

缺点：不够相信，不够全面

## 延伸及扩展问题回答参考

**问题**：目前前端对node的应用场景有哪些？

解答： 前端自动化，客户端服务都在大规模的使用node进行问题的解决！

## 。项目中体现经验的点

**可以解决前端本地开发的服务问题，可以实现接口的代理，可以实现客户端服务等**

## 论坛参考

<https://blog.csdn.net/hsj1669666567/article/details/79568545>

https://cnodejs.org/topic/53c88a83c9507b4044b0a78a

# 谈谈Vuejs？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 95% | 3星 | Es6 | 异步 | Es6异步操作 |

## 通常解法

vue生命周期一共有11个，

**beforeCreate,created**

**beforeMount,mounted**

**beforeUpdate, updated**

**actived** , deactivated

beforeDestroy, destroyed

errorCaptured

## 通用大牛级解法

vue的生命周期主要分为几个简单，数据初始化，dom挂载，数据更新，组件卸载，在一个就是开启了组件缓存的时候，会有组件启用和组件停用阶段，每个阶段都去前后两个钩子除了缓存的那俩

数据初始化阶段

beforeCreate：在实例初始化之后，数据观测 (data observer) 和 event/watcher 事件配置之前被调用。

created：实例已经创建完成之后被调用。在这一步，实例已完成以下的配置：数据观测 (data observer)，属性和方法的运算，watch/event 事件回调。然而，挂载阶段还没开始，$el 属性目前不可见。

dom挂载阶段

beforeMount：在挂载开始之前被调用：相关的 render 函数首次被调用。

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。

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 替换掉 mounted

数据跟新阶段

beforeUpdate：数据更新时调用，发生在虚拟 DOM 重新渲染和打补丁之前。

updated：由于数据更改导致的虚拟 DOM 重新渲染和打补丁，在这之后会调用该钩子。

当这个钩子被调用时，组件 DOM 已经更新，所以你现在可以执行依赖于 DOM 的操作。然而在大多数情况下，你应该避免在此期间更改状态。如果要相应状态改变，通常最好使用计算属性或 watcher 取而代之。

updated 不会承诺所有的子组件也都一起被重绘。如果你希望等到整个视图都重绘完毕，可以用 vm.$nextTick 替换掉 updated：

缓存启用的时候会有下面两个钩子

activated：keep-alive 组件激活时调用。

deactivated：keep-alive 组件停用时调用。

组件卸载的时候：

beforeDestroy：实例销毁之前调用。在这一步，实例仍然完全可用。

destroyed：Vue 实例销毁后调用。调用后，Vue 实例指示的所有东西都会解绑定，所有的事件监听器会被移除，所有的子实例也会被销毁。

## 解法对比及优缺点

**综上所述**：第一种只是说出了生命周期方法，而第二种说出生命周期的执行作用

## 延伸及扩展问题回答参考

**问题**： 数据请求应该在那个生命周期，

**解答**：在created里面

## 项目中体现经验的点

组件中必须的内容

## 论坛参考

<https://cn.vuejs.org/v2/api>

# 说一下keep-alive？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 90% | 5星 | Javascript | 数据存储 | 存储 |

## 通常解法

Vuex 是适用于 Vue.js 应用的状态管理库，为应用中的所有组件提供集中式的状态存储与操作，保证了所有状态以可预测的方式进行修改；

state: state 定义了应用状态的数据结构，同样可以在这里设置默认的初始状态。

actions:Actions 即是定义提交触发更改信息的描述，常见的例子有从服务端获取数据，在数据获取完成后会调用store.commit()来调用更改 Store 中的状态。可以在组件中使用dispatch来发出 Actions。

mutations: 调用 mutations 是唯一允许更新应用状态的地方。

getters: Getters 允许组件从 Store 中获取数据，譬如我们可以从 Store 中的 projectList 中筛选出已完成的项目列表

modules: modules 对象允许将单一的 Store 拆分为多个 Store 的同时保存在单一的状态树中。随着应用复杂度的增加，这种拆分能够更好地组织代码

但是vuex也有缺点就是，vuex中保存的数据是和网页的生命周期同步的，当执行页面刷新的时候vuex中所有数据都会消失复位到初始状态，所以不太适合做有分享页面的数据交互（在这种项目中vuex只适合数据的集中管理，不适合数据的存储，这种情况一般是使用路由传递参数会好一些），适合后台管理系统多一些，后台管理系统一般都是公司内部使用；

## 通用大牛级解法

Vuex 是专门针对vue开发的应用状态管理库，为应用中的所有组件提供集中式的状态存储与操作，保证了所有状态以可预测的方式进行修改；

state: state 定义了应用状态的数据结构，同样可以在这里设置默认的初始状态。

actions:Actions 即是定义提交触发更改信息的描述，常见的例子有从服务端获取数据，在数据获取完成后会调用store.commit()来调用更改 Store 中的状态。可以在组件中使用dispatch来发出 Actions。

mutations: 调用 mutations 是唯一允许更新应用状态的地方。

getters: Getters 允许组件从 Store 中获取数据，譬如我们可以从 Store 中的 projectList 中筛选出已完成的项目列表

modules: modules 对象允许将单一的 Store 拆分为多个 Store 的同时保存在单一的状态树中。随着应用复杂度的增加，这种拆分能够更好地组织代码

在项目vue中主要使用vuex需要注意的是vuex相当于一个全局的状态，所以不要在vuex保存大量的数据，只需要保存全局通用数据就可以了，还有就是当刷新页面的时候由于是js对象所以对象中的数据都会回归初始化，所以一般都需要配合本地存储一块使用，在就是vuex针对于vue的数据检测做了适配，所以在vue中使用vuex做状态管理还是比较不错的

## 解法对比及优缺点

通常解法，vuex的规则解释。通用级解法，说出了vuex的特性，优缺点，以及

。

## 项目中体现经验的点

登录判断，本地数据线上数据的同步

## 论坛参考

<https://vuex.vuejs.org/>

# 谈谈vue脚手架？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 50% | 3星 | 小程序 | 功能抽象 | 组件 |

## 通常解法

从小程序基础库版本 1.6.3 开始，小程序支持简洁的组件化编程。所有自定义组件相关特性都需要基础库版本 1.6.3 或更高。

开发者可以将页面内的功能模块抽象成自定义组件，以便在不同的页面中重复使用；也可以将复杂的页面拆分成多个低耦合的模块，有助于代码维护。自定义组件在使用时与基础组件非常相似。

在自定义组件的 js 文件中，需要使用 Component() 来注册组件，并提供组件的属性定义、内部数据和自定义方法。

组件的属性值和内部数据将被用于组件 wxml 的渲染，其中，属性值是可由组件外部传入的

组件构造器包含的属性和方法：

properties，data，methods，behaviors（mixin），created，attached，ready，moved，detached，relations等等

## 通用大牛级解法

理解小程序组件可以对比vue组件或者react组件，页面中多组件嵌套肯定会涉及跨组件传值，那么小程序的组件传值是如何实现的。Vue或者react都属于单向数据流的数据管理方式，小程序组件也是类似的。通过在组件上添加自定义属性，属性值可以是父组件的任意变量，在组建内使用properties属性来接收。例如：<test data=”{{count}}” bindtest=”func”></test>

Component({

Properties:{

data:{

Type:String,

Value:’’,

Observer:function(){}

}

}

})

这是父子组件传值，其中type限制组件接受值的格式，value可以设置该属性对应的默认值，observer类似于vue的watch，可以订阅属性值的变化，进而做相应的操作。

小程序组件由子组件向父组件传值则用的的事件派发的方式通过triggerEvent方法触发，并在组件标签上bind事件监听，执行回调函数，来实现子父传值。类似于vue的$emit, v-on组合

除了这些传值方式以外，还有全局对象挂载的方式。任何页面都可以通过getApp()得到app实例，该app实例是小程序的全局对象，拥有上帝视角，并且可以添加自定义属性，利用这点也能实现跨组件传值，不再限于父子组件之间。

## 解法对比及优缺点

* 普通解法只是小程序组件的概念和api，以及基本的使用。
* 大牛级解法对小程序组件的数据流进行了梳理，解释清楚了实际操作中会遇到的具体问题，以及解决方式，体现了丰富的经验，以及数据流概念

## 延伸及扩展问题回答参考

对比vue，react，小程序组件各自的特点，总结它们共性和不同点

## 项目中体现经验的点

深刻理解小程序组件的特点和使用方法；

## 论坛参考

## <https://developers.weixin.qq.com/miniprogram/dev/framework/custom-component/events.html>

# 谈谈对node对的看法？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 70% | 4星 | 前端 | 数据安全 | 数据安全 |

## 通常解法

Node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行环境。   
Node.js 使用了一个事件驱动、非阻塞式 I/O 的模型，使其轻量又高效。   
Node.js 的包管理器 npm，是全球最大的开源库生态系统。

Node解决了js跑在服务端的一个空缺

## 通用大牛级解法

加上通用解法后

可以解决高并发，它是单线程，当访问量很多时，将访问者分配到不同的内存中，不同的内存区做不同的事，以快速解决这个线程。就像医院的分科室看病人。效率快，但消耗内存大、异步和事件驱动。概扩起来就三点：单线程、异步I/O、事件驱动。nodejs离不开ChormeV8引擎，也就是V8引擎是来解释javascript。用nodejs来搭建高性能的Web服务器，因此node.js是基于服务器端的javascript

node主要应用场景是在大前端，阿里的思路是比较合适的，但是必须要注意，绝对不能让node做太多的业务逻辑，他只适合接受人家生成好的数据，然后或渲染后，或直接发送到客户端。如果让node做复杂的业务逻辑，那会得不偿失的

回调模式下的异步是有明显缺陷的，程序的执行顺序必须依靠回调来保证，没有层层回调，就没有可以保障的逻辑顺序，这也就注定了，node不能做复杂的业务逻辑。javascript语言本身也一直在和回调做斗争，promise，generator都可以将回调包装起来，在代码的某个部分形成形式同步，但是这种模式进化的还不完全，还不能做到与回调完全割裂，做到完全的形式同步。但是形式同步肯定是发展的方向，这种模式即可以获得异步的好处，又可以有效回避回调带来的编程困难，在业务逻辑上可以更简单的表达。

就现在的环境来说，大家的思路还没转过弯，对回调的批评认为都是不好的，这些人是不敢面对现实，javascript都在变，这些人的脑子却不肯变，还以为回调就代表异步。

Node.js的发展给javascript注入了新的生命力

至于javascript的垢病，个人感觉，不在它的callback而在它的随意性，随意到想怎么写都行，但正是这点给它带来了惊人的开发效率，做好代码规范和文档工作可以减少javascript的随意性带来的负面影响。

WEB端、移动端、桌面端、甚至嵌入式，javascript已经无处不在。接下来，ES6的实现会让众多习惯同步或者不喜欢回调的开发人员能够更快地上手javascript写出符合他们思维习惯的代码，这些开发人员会是更大的群体，那么也许javascript会横扫应用开发也不一定。

所以，javascript很有前途，那Node.js自然就有前途。

## 解法对比及优缺点

大牛级解法：

优点：可以比较深入的解释清楚node底层原理和带来的一些良性循环；

缺点：如果对基础知识不牢固，可能会带来一系列的问题；

通用解法：

优点：可以比较概括的回答面试的问题

缺点：不够相信，不够全面

## 延伸及扩展问题回答参考

**问题**：目前前端对node的应用场景有哪些？

解答： 前端自动化，客户端服务都在大规模的使用node进行问题的解决！

## 。项目中体现经验的点

**可以解决前端本地开发的服务问题，可以实现接口的代理，可以实现客户端服务等**

## 论坛参考

<https://blog.csdn.net/hsj1669666567/article/details/79568545>

https://cnodejs.org/topic/53c88a83c9507b4044b0a78a

# Vue的脚手架？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 95% | 3星 | Es6 | 异步 | Es6异步操作 |

## 通常解法

vue生命周期一共有11个，

**beforeCreate,created**

**beforeMount,mounted**

**beforeUpdate, updated**

**actived** , deactivated

beforeDestroy, destroyed

errorCaptured

## 通用大牛级解法

vue的生命周期主要分为几个简单，数据初始化，dom挂载，数据更新，组件卸载，在一个就是开启了组件缓存的时候，会有组件启用和组件停用阶段，每个阶段都去前后两个钩子除了缓存的那俩

数据初始化阶段

beforeCreate：在实例初始化之后，数据观测 (data observer) 和 event/watcher 事件配置之前被调用。

created：实例已经创建完成之后被调用。在这一步，实例已完成以下的配置：数据观测 (data observer)，属性和方法的运算，watch/event 事件回调。然而，挂载阶段还没开始，$el 属性目前不可见。

dom挂载阶段

beforeMount：在挂载开始之前被调用：相关的 render 函数首次被调用。

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。

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 替换掉 mounted

数据跟新阶段

beforeUpdate：数据更新时调用，发生在虚拟 DOM 重新渲染和打补丁之前。

updated：由于数据更改导致的虚拟 DOM 重新渲染和打补丁，在这之后会调用该钩子。

当这个钩子被调用时，组件 DOM 已经更新，所以你现在可以执行依赖于 DOM 的操作。然而在大多数情况下，你应该避免在此期间更改状态。如果要相应状态改变，通常最好使用计算属性或 watcher 取而代之。

updated 不会承诺所有的子组件也都一起被重绘。如果你希望等到整个视图都重绘完毕，可以用 vm.$nextTick 替换掉 updated：

缓存启用的时候会有下面两个钩子

activated：keep-alive 组件激活时调用。

deactivated：keep-alive 组件停用时调用。

组件卸载的时候：

beforeDestroy：实例销毁之前调用。在这一步，实例仍然完全可用。

destroyed：Vue 实例销毁后调用。调用后，Vue 实例指示的所有东西都会解绑定，所有的事件监听器会被移除，所有的子实例也会被销毁。

## 解法对比及优缺点

**综上所述**：第一种只是说出了生命周期方法，而第二种说出生命周期的执行作用

## 延伸及扩展问题回答参考

**问题**： 数据请求应该在那个生命周期，

**解答**：在created里面

## 项目中体现经验的点

组件中必须的内容

## 论坛参考

<https://cn.vuejs.org/v2/api>

# 说一下seo？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 90% | 5星 | Javascript | 数据存储 | 存储 |

## 通常解法

Vuex 是适用于 Vue.js 应用的状态管理库，为应用中的所有组件提供集中式的状态存储与操作，保证了所有状态以可预测的方式进行修改；

state: state 定义了应用状态的数据结构，同样可以在这里设置默认的初始状态。

actions:Actions 即是定义提交触发更改信息的描述，常见的例子有从服务端获取数据，在数据获取完成后会调用store.commit()来调用更改 Store 中的状态。可以在组件中使用dispatch来发出 Actions。

mutations: 调用 mutations 是唯一允许更新应用状态的地方。

getters: Getters 允许组件从 Store 中获取数据，譬如我们可以从 Store 中的 projectList 中筛选出已完成的项目列表

modules: modules 对象允许将单一的 Store 拆分为多个 Store 的同时保存在单一的状态树中。随着应用复杂度的增加，这种拆分能够更好地组织代码

但是vuex也有缺点就是，vuex中保存的数据是和网页的生命周期同步的，当执行页面刷新的时候vuex中所有数据都会消失复位到初始状态，所以不太适合做有分享页面的数据交互（在这种项目中vuex只适合数据的集中管理，不适合数据的存储，这种情况一般是使用路由传递参数会好一些），适合后台管理系统多一些，后台管理系统一般都是公司内部使用；

## 通用大牛级解法

Vuex 是专门针对vue开发的应用状态管理库，为应用中的所有组件提供集中式的状态存储与操作，保证了所有状态以可预测的方式进行修改；

state: state 定义了应用状态的数据结构，同样可以在这里设置默认的初始状态。

actions:Actions 即是定义提交触发更改信息的描述，常见的例子有从服务端获取数据，在数据获取完成后会调用store.commit()来调用更改 Store 中的状态。可以在组件中使用dispatch来发出 Actions。

mutations: 调用 mutations 是唯一允许更新应用状态的地方。

getters: Getters 允许组件从 Store 中获取数据，譬如我们可以从 Store 中的 projectList 中筛选出已完成的项目列表

modules: modules 对象允许将单一的 Store 拆分为多个 Store 的同时保存在单一的状态树中。随着应用复杂度的增加，这种拆分能够更好地组织代码

在项目vue中主要使用vuex需要注意的是vuex相当于一个全局的状态，所以不要在vuex保存大量的数据，只需要保存全局通用数据就可以了，还有就是当刷新页面的时候由于是js对象所以对象中的数据都会回归初始化，所以一般都需要配合本地存储一块使用，在就是vuex针对于vue的数据检测做了适配，所以在vue中使用vuex做状态管理还是比较不错的

## 解法对比及优缺点

通常解法，vuex的规则解释。通用级解法，说出了vuex的特性，优缺点，以及

。

## 项目中体现经验的点

登录判断，本地数据线上数据的同步

## 论坛参考

<https://vuex.vuejs.org/>

# 谈谈对vuexcookie对的看法？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 70% | 4星 | 前端 | 数据安全 | 数据安全 |

## 通常解法

Node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行环境。   
Node.js 使用了一个事件驱动、非阻塞式 I/O 的模型，使其轻量又高效。   
Node.js 的包管理器 npm，是全球最大的开源库生态系统。

Node解决了js跑在服务端的一个空缺

## 通用大牛级解法

加上通用解法后

可以解决高并发，它是单线程，当访问量很多时，将访问者分配到不同的内存中，不同的内存区做不同的事，以快速解决这个线程。就像医院的分科室看病人。效率快，但消耗内存大、异步和事件驱动。概扩起来就三点：单线程、异步I/O、事件驱动。nodejs离不开ChormeV8引擎，也就是V8引擎是来解释javascript。用nodejs来搭建高性能的Web服务器，因此node.js是基于服务器端的javascript

node主要应用场景是在大前端，阿里的思路是比较合适的，但是必须要注意，绝对不能让node做太多的业务逻辑，他只适合接受人家生成好的数据，然后或渲染后，或直接发送到客户端。如果让node做复杂的业务逻辑，那会得不偿失的

回调模式下的异步是有明显缺陷的，程序的执行顺序必须依靠回调来保证，没有层层回调，就没有可以保障的逻辑顺序，这也就注定了，node不能做复杂的业务逻辑。javascript语言本身也一直在和回调做斗争，promise，generator都可以将回调包装起来，在代码的某个部分形成形式同步，但是这种模式进化的还不完全，还不能做到与回调完全割裂，做到完全的形式同步。但是形式同步肯定是发展的方向，这种模式即可以获得异步的好处，又可以有效回避回调带来的编程困难，在业务逻辑上可以更简单的表达。

就现在的环境来说，大家的思路还没转过弯，对回调的批评认为都是不好的，这些人是不敢面对现实，javascript都在变，这些人的脑子却不肯变，还以为回调就代表异步。

Node.js的发展给javascript注入了新的生命力

至于javascript的垢病，个人感觉，不在它的callback而在它的随意性，随意到想怎么写都行，但正是这点给它带来了惊人的开发效率，做好代码规范和文档工作可以减少javascript的随意性带来的负面影响。

WEB端、移动端、桌面端、甚至嵌入式，javascript已经无处不在。接下来，ES6的实现会让众多习惯同步或者不喜欢回调的开发人员能够更快地上手javascript写出符合他们思维习惯的代码，这些开发人员会是更大的群体，那么也许javascript会横扫应用开发也不一定。

所以，javascript很有前途，那Node.js自然就有前途。

## 解法对比及优缺点

大牛级解法：

优点：可以比较深入的解释清楚node底层原理和带来的一些良性循环；

缺点：如果对基础知识不牢固，可能会带来一系列的问题；

通用解法：

优点：可以比较概括的回答面试的问题

缺点：不够相信，不够全面

## 延伸及扩展问题回答参考

**问题**：目前前端对node的应用场景有哪些？

解答： 前端自动化，客户端服务都在大规模的使用node进行问题的解决！

## 。项目中体现经验的点

**可以解决前端本地开发的服务问题，可以实现接口的代理，可以实现客户端服务等**

## 论坛参考

<https://blog.csdn.net/hsj1669666567/article/details/79568545>

https://cnodejs.org/topic/53c88a83c9507b4044b0a78a

# 说一下promise？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 95% | 3星 | Es6 | 异步 | Es6异步操作 |

## 通常解法

vue生命周期一共有11个，

**beforeCreate,created**

**beforeMount,mounted**

**beforeUpdate, updated**

**actived** , deactivated

beforeDestroy, destroyed

errorCaptured

## 通用大牛级解法

vue的生命周期主要分为几个简单，数据初始化，dom挂载，数据更新，组件卸载，在一个就是开启了组件缓存的时候，会有组件启用和组件停用阶段，每个阶段都去前后两个钩子除了缓存的那俩

数据初始化阶段

beforeCreate：在实例初始化之后，数据观测 (data observer) 和 event/watcher 事件配置之前被调用。

created：实例已经创建完成之后被调用。在这一步，实例已完成以下的配置：数据观测 (data observer)，属性和方法的运算，watch/event 事件回调。然而，挂载阶段还没开始，$el 属性目前不可见。

dom挂载阶段

beforeMount：在挂载开始之前被调用：相关的 render 函数首次被调用。

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。

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 替换掉 mounted

数据跟新阶段

beforeUpdate：数据更新时调用，发生在虚拟 DOM 重新渲染和打补丁之前。

updated：由于数据更改导致的虚拟 DOM 重新渲染和打补丁，在这之后会调用该钩子。

当这个钩子被调用时，组件 DOM 已经更新，所以你现在可以执行依赖于 DOM 的操作。然而在大多数情况下，你应该避免在此期间更改状态。如果要相应状态改变，通常最好使用计算属性或 watcher 取而代之。

updated 不会承诺所有的子组件也都一起被重绘。如果你希望等到整个视图都重绘完毕，可以用 vm.$nextTick 替换掉 updated：

缓存启用的时候会有下面两个钩子

activated：keep-alive 组件激活时调用。

deactivated：keep-alive 组件停用时调用。

组件卸载的时候：

beforeDestroy：实例销毁之前调用。在这一步，实例仍然完全可用。

destroyed：Vue 实例销毁后调用。调用后，Vue 实例指示的所有东西都会解绑定，所有的事件监听器会被移除，所有的子实例也会被销毁。

## 解法对比及优缺点

**综上所述**：第一种只是说出了生命周期方法，而第二种说出生命周期的执行作用

## 延伸及扩展问题回答参考

**问题**： 数据请求应该在那个生命周期，

**解答**：在created里面

## 项目中体现经验的点

组件中必须的内容

## 论坛参考

<https://cn.vuejs.org/v2/api>

# 说一下jsonp？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 面试概率 | 级别 | 应用模块或方向 | 解决问题 | 考核点 |
| 90% | 5星 | Javascript | 数据存储 | 存储 |

## 通常解法

Vuex 是适用于 Vue.js 应用的状态管理库，为应用中的所有组件提供集中式的状态存储与操作，保证了所有状态以可预测的方式进行修改；

state: state 定义了应用状态的数据结构，同样可以在这里设置默认的初始状态。

actions:Actions 即是定义提交触发更改信息的描述，常见的例子有从服务端获取数据，在数据获取完成后会调用store.commit()来调用更改 Store 中的状态。可以在组件中使用dispatch来发出 Actions。

mutations: 调用 mutations 是唯一允许更新应用状态的地方。

getters: Getters 允许组件从 Store 中获取数据，譬如我们可以从 Store 中的 projectList 中筛选出已完成的项目列表

modules: modules 对象允许将单一的 Store 拆分为多个 Store 的同时保存在单一的状态树中。随着应用复杂度的增加，这种拆分能够更好地组织代码

但是vuex也有缺点就是，vuex中保存的数据是和网页的生命周期同步的，当执行页面刷新的时候vuex中所有数据都会消失复位到初始状态，所以不太适合做有分享页面的数据交互（在这种项目中vuex只适合数据的集中管理，不适合数据的存储，这种情况一般是使用路由传递参数会好一些），适合后台管理系统多一些，后台管理系统一般都是公司内部使用；

## 通用大牛级解法

Vuex 是专门针对vue开发的应用状态管理库，为应用中的所有组件提供集中式的状态存储与操作，保证了所有状态以可预测的方式进行修改；

state: state 定义了应用状态的数据结构，同样可以在这里设置默认的初始状态。

actions:Actions 即是定义提交触发更改信息的描述，常见的例子有从服务端获取数据，在数据获取完成后会调用store.commit()来调用更改 Store 中的状态。可以在组件中使用dispatch来发出 Actions。

mutations: 调用 mutations 是唯一允许更新应用状态的地方。

getters: Getters 允许组件从 Store 中获取数据，譬如我们可以从 Store 中的 projectList 中筛选出已完成的项目列表

modules: modules 对象允许将单一的 Store 拆分为多个 Store 的同时保存在单一的状态树中。随着应用复杂度的增加，这种拆分能够更好地组织代码

在项目vue中主要使用vuex需要注意的是vuex相当于一个全局的状态，所以不要在vuex保存大量的数据，只需要保存全局通用数据就可以了，还有就是当刷新页面的时候由于是js对象所以对象中的数据都会回归初始化，所以一般都需要配合本地存储一块使用，在就是vuex针对于vue的数据检测做了适配，所以在vue中使用vuex做状态管理还是比较不错的

## 解法对比及优缺点

通常解法，vuex的规则解释。通用级解法，说出了vuex的特性，优缺点，以及

。

## 项目中体现经验的点

登录判断，本地数据线上数据的同步

## 论坛参考

<https://vuex.vuejs.org/>