**装饰器与高阶组件**

1. 装饰器

其实只是一个语法糖. 就是以一种更为简洁的方式来来包装修改类的行为。装饰对象包括：类、类的属性、类的方法。装饰器是一种函数，写成@ + 函数名。它可以放在类和类方法的定义前面。

装饰器对类的行为的改变，是代码编译时发生的，而不是在运行时。这意味着，装饰器能在编译阶段运行代码。也就是说，装饰器本质就是编译时执行的函数。

如果同一个方法有多个装饰器，会像剥洋葱一样，先从外到内进入，然后由内向外执行。

装饰器只能用于类和类的方法，不能用于函数，因为存在函数提升。

2.高阶组件  
 简单来说，高阶组件是一个函数，能够接受一个组件并返回一个新的组件。组件是将 props 转化成 UI ，然而 高阶组件将一个组价转化成另外一个组件。

**Hooks简介和概述？**

Hooks 是 React 函数组件内一类特殊的函数（通常以 "use" 开头，比如 "useState"），使开发者能够在 function component 里依旧使用 state 和 life-cycles，以及使用 custom hooks 复用业务逻辑。

**浏览器缓存**

浏览器缓存过程：强缓存，协商缓存。

浏览器缓存位置一般分为四类：

Service Worker(客户端和服务器之间的一个代理服务器)

Memory Cache(内存中的缓存)

Disk Cache(磁盘中的缓存)

Push Cache(一旦页面关闭就被释放,优先级最低)。

存存储图像和网页等资源主要缓存在disk cache，操作系统缓存文件等资源大部分都会缓存在memory cache中  
协商缓存

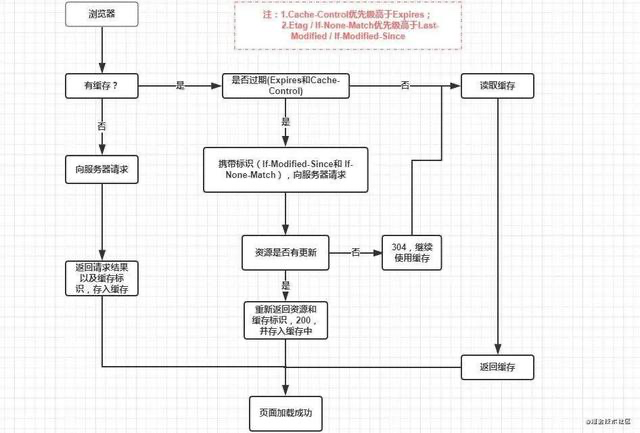
Last-Modified / If-Modified-Since

Etag / If-None-Match

* 精确度上Etag要优于Last-Modified
* 优先级上:服务器校验优先考虑Etag
* 性能上:Etag逊色于Last-Modified

协商缓存生效，返回304

协商缓存失效，返回200和请求结果





4. 缓存方案

目前的项目大多使用这种缓存方案的：

HTML: 协商缓存；

css、js、图片：强缓存，文件名带上hash。

强缓存与协商缓存的区别

1. 强缓存不发请求到服务器，所以有时候资源更新了浏览器还不知道，但是协商缓存会发请求到服务器，所以资源是否更新，服务器肯定知道。

2. 大部分web服务器都默认开启协商缓存。

刷新对于强缓存和协商缓存的影响

1. 当ctrl+f5强制刷新网页时，直接从服务器加载，跳过强缓存和协商缓存。

2. 当f5刷新网页时，跳过强缓存，但是会检查协商缓存。

3. 浏览器地址栏中写入URL，回车 浏览器发现缓存中有这个文件了，不用继续请求了，直接去缓存拿。（最快）

# **[localstorage sessionstorage和cookie的区别](https://www.cnblogs.com/candy-xia/p/11561542.html)**

三者区别

1）存储大小

cookie：一般不超过4K（因为每次http请求都会携带cookie、所以cookie只适合保存很小的数据，如会话标识）

sessionStorage：5M或者更大

localStorage：5M或者更大

2）数据有效期

cookie：一般由服务器生成，可以设置失效时间；若没有设置时间，关闭浏览器cookie失效，若设置了时间，cookie就会存放在硬盘里，过期才失效

sessionStorage：仅当前浏览器窗口关闭之前有效，关闭页面或者浏览器会被清除

localStorage：永久有效，窗口或者浏览器关闭也会一直保存，除非手动永久清除，因此用作持久数据

3）作用域

cookie：在所有同源窗口中都是共享的

sessionStorage：在同一个浏览器窗口是共享的（不同浏览器、同一个页面也是不共享的）

localStorage：在所有同源窗口中都是共享的

4）通信

ccokie：十种携带在同源的http请求中，即使不需要，故cookie在浏览器和服务器之间来回传递；如果使用cookie保存过多数据会造成性能问题

sessionStorage：仅在客户端（即浏览器）中保存，不参与和服务器的通信；不会自动把数据发送给服务器，仅在本地保存

localStorage：仅在客户端（即浏览器）中保存，不参与和服务器的通信；不会自动把数据发送给服务器，仅在本地保存

5）易用性

cookie：需要自己进行封装，原生的cookie接口不够友好

sessionStorage：原生接口可以接受，可以封装来对Object和Array有更好的支持

localStorage：原生接口可以接受，可以封装来对Object和Array有更好的支持

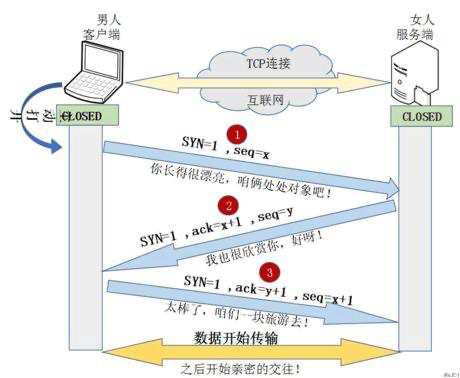
6）应用场景

cookie：判断用户是否登录过网站，以便实现下次自动登录或记住密码；保存事件信息等

sessionStorage：敏感账号一次性登录；单页面用的较多（sessionStorage 可以保证打开页面时 sessionStorage 的数据为空）

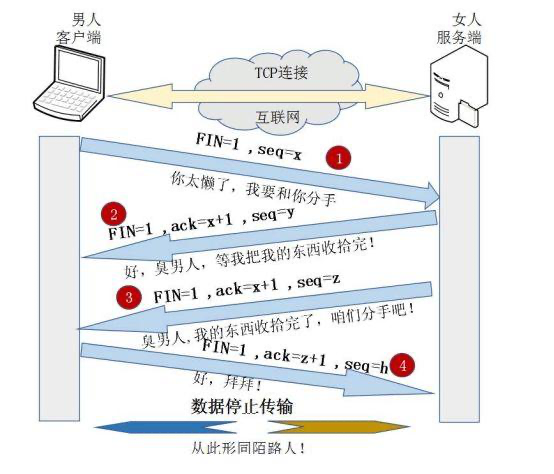
localStorage：常用于长期登录（判断用户是否已登录），适合长期保存在本地的数据

# **TCP 连接的“ 三次握手 ”与“ 四次挥手 ”**



为了防止服务器端开启一些无用的连接增加服务器开销以及防止已失效的连接请求报文段突然又传送到了服务端，因而产生错误。

“第三次握手”是客户端向服务器端发送数据，这个数据就是要告诉服务器，客户端有没有收到服务器“第二次握手”时传过去的数据。若发送的这个数据是“收到了”的信息，接收后服务器就正常建立TCP连接，否则建立TCP连接失败，服务器关闭连接端口。由此减少服务器开销和接收到失效请求发生的错误。



TCP释放连接时之所以需要“四次挥手”,是因为FIN释放连接报文与ACK确认接收报文是分别由第二次和第三次"握手"传输的。

建立连接时，被动方服务器端结束CLOSED阶段进入“握手”阶段并不需要任何准备，可以直接返回SYN和ACK报文，开始建立连接。释放连接时，被动方服务器，突然收到主动方客户端释放连接的请求时并不能立即释放连接，因为还有必要的数据需要处理，所以服务器先返回ACK确认收到报文，经过CLOSE-WAIT阶段准备好释放连接之后，才能返回FIN释放连接报文。

**国际化 react-intl**

首先解决静态国际化，即根据浏览器的语言，自动加载对应的语言模板。这里只需判断navigator.language类型便可，而后经过react-intl提供了IntlProvider组件，加载组件属性的locale和messages，最后在须要用到国际化的组件里，引入FormattedMessage组件（react-intl内置），经过id映射到对应的国际化文件里的属性（例以下面的en\_US.js的hello）。便可实现静态国际化。r

# **vuex与redux**

vuex仅仅是作为vue的一个插件而存在，每一个vue插件都需要有一个公开的install方法,调用了一下applyMixin方法，该方法主要作用就是在所有组件的beforeCreate生命周期注入了设置this.$store这样一个对象，

其本质就是将我们传入的state作为一个隐藏的vue组件的data,也就是说，我们的commit操作，本质上其实是修改这个组件的data值，在利用vue的响应式完成更新 , redux 是将整个应用的state存储在一个公共的store文件当中，组件可以通过分发（dispatch）一个动作或者是行为（action）给这个公用的store，而不是直接去通知其他组件，组件内部通过订阅store中的状态state来刷新自己的视图。这里我个人对的理解是，在我们的组件内部有个类似于监听器的东西，一旦监听到store中的值发生了改变就会刷新我们的页面。

redux 三大原则

唯一数据源

保持只读状态

数据改变只能通过纯函数来执行

redux更新视图

用<Provider store={store}><Provider>包裹需要暴露store的子组件,通过store在父子组件之间共享状态,通过store.subscribe监听状态改变，更新界面，通过store.getState()获取状态。

为什么 Redux 中的 reducer 和 vuex 中的 mutation 都是同步的状态更新？

就是为了在开发工具里面好跟踪状态变化，因而把它设计为同步的,vuex和redux都是一种状态管理机制。然后他们会有自己的state（状态）和修改state的方法，修改state的方法涉及到同步和异步，vuex的处理方式是同步在mutation里面，异步在actions里面，然后redux的同步就是reducer,异步更多的是用户自己去通过中间件的方式去实现

vuex是专门为Vuejs应用程序设计的状态管理工具

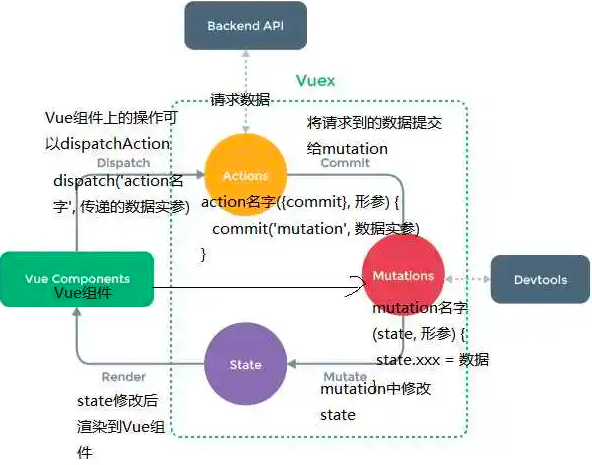
state:是存储的基本数据。

mutations:提交更改数据。

getter:对state加工，和computed计算属性一样。

actions：处理异步，通过store.commit方法触发mutations中的方法，从而改变state值。

module：是store分割的模块，每个模块拥有自己的state、mutations、getters、actions。



vuex的流向：

view——>commit——>mutations——>state变化——>view变化（同步操作）

view——>dispatch——>actions——>mutations——>state变化——>view变化（异步操作）

redux的流向：view——>actions——>reducer——>state变化——>view变化（同步异步一样）

Redux工作流程：

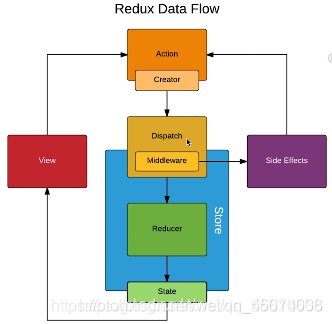
1.视图组件在 redux 中派发 action 方法

2.action 通过 store 的 dispatch 方法派发给 store

3.store 接收 action，连同之前的 state，一起传递给 reducer ，reducer 实现更新 state 的动作后返回新的 state 给 store

4.store 去改变自己的 state

5.视图组件连接了 store ，store 中的 state 更改后，视图组件的数据也就随之更改从而重新渲染页面。



[#](https://link.zhihu.com/?target=https://lq782655835.github.io/blogs/react/diff-vuex-redux.html%23%E4%B8%8D%E5%90%8C%E7%82%B9" \t "/Users/admin/Documents\\x/_blank)不同点

从实现原理上来说：

Redux 使用的是不可变数据，而Vuex的数据是可变的。Redux每次都是用新的state替换旧的state，而Vuex是直接修改

Redux 在检测数据变化的时候，是通过 diff 的方式比较差异的，而Vuex其实和Vue的原理一样，是通过 getter/setter来比较的

从表现层来说：

vuex定义了state、getter、mutation、action四个对象；redux定义了state、reducer、action。

vuex中state统一存放，方便理解；reduxstate依赖所有reducer的初始值

vuex有getter,目的是快捷得到state；redux没有这层，react-redux mapStateToProps参数做了这个工作。

vuex中mutation只是单纯赋值(很浅的一层)；redux中reducer只是单纯设置新state(很浅的一层)。他俩作用类似，但书写方式不同

vuex中action有较为复杂的异步ajax请求；redux中action中可简单可复杂,简单就直接发送数据对象（{type:xxx, your-data}）,复杂需要调用异步ajax（依赖redux-thunk插件）。

vuex触发方式有两种commit同步和dispatch异步；redux同步和异步都使用dispatch

**webpack**

loader，webpack自身只支持js和json这两种格式的文件，对于其他文件需要通过loader将其转换为commonJS规范的文件后，webpack才能解析到

plugin是一个扩展器，它丰富了webpack本身，针对是loader结束后，webpack打包的整个过程，它并不直接操作文件，而是基于事件机制工作，会监听webpack打包过程中的某些节点，执行广泛的任务

构建流程:

webpack 的运行流程是一个串行的过程，从启动到结束会依次执行以下流程：

初始化参数：从配置文件和 Shell 语句中读取与合并参数，得出最终的参数；

开始编译：用上一步得到的参数初始化 Compiler 对象，加载所有配置的插件，执行对象的 run 方法开始执行编译；

确定入口：根据配置中的 entry 找出所有的入口文件

编译模块：从入口文件出发，调用所有配置的 Loader 对模块进行翻译，再找出该模块依赖的模块，再递归本步骤直到所有入口依赖的文件都经过了本步骤的处理；

完成模块编译：在经过第4步使用 Loader 翻译完所有模块后，得到了每个模块被翻译后的最终内容以及它们之间的依赖关系；

输出资源：根据入口和模块之间的依赖关系，组装成一个个包含多个模块的 Chunk，再把每个 Chunk 转换成一个单独的文件加入到输出列表，这步是可以修改输出内容的最后机会；

输出完成：在确定好输出内容后，根据配置确定输出的路径和文件名，把文件内容写入到文件系统。

在以上过程中，webpack 会在特定的时间点广播出特定的事件，插件在监听到感兴趣的事件后会执行特定的逻辑，并且插件可以调用 webpack 提供的 API 改变 webpack 的运行结果。

对于应用使用 webpack，对于类库使用 Rollup

webpack 基于入口使用loader处理文件和plugin进行罗占,拆分代码， 按需加载；

Rollup基于任务和流,找到一个文件,做链式操作,更新所有资源放在同一个地方，一次性加载

vite：实时的打包，开始不会进行打包，所以启动非常快，直接请求所需模块并实时编译

**vue优化**

1. 路由懒加载
2. keep-alive缓存页面
3. v-show/v-if注意 computer/watcher
4. v-if/v-for避免同时使用 v-for添加唯一key
5. 长列表冻结不添加响应式 object.freeze()
6. 长列表虚拟滚动 react-window
7. 事件销毁
8. 图片懒加载 v-lazy
9. 第三方插件按需引入
10. 无状态组件标记为函数式组件
11. 子组件分割
12. 服务端渲染ssr seo
13. GZIP 资源CDN 浏览器缓存

**Vue实现双向数据绑定**

主要采用：数据劫持结合“发布-订阅”模式的方式，通Object.defineProperty（）的 set 和 get，在数据变动时发布消息给订阅者触发监听。

当一个 Vue 实例创建时 会遍历 data 选项，Vue 将遍历此对象所有的 property，并用 Object.defineProperty 把这些 property 全部转为 getter/setter。每个组件实例都对应一个 watcher 实例，它会在组件渲染的过程中把“接触”过的数据 property 记录为依赖。之后当依赖项的 setter 触发时，会通知 watcher，从而使它关联的组件重新渲染。

**Vue实现步骤**

1. Vue类负责把data中的属性注入到Vue实例，并调用Observer类和Compiler类。

2. Observer类负责数据劫持，把每一个data转换成getter和setter。其核心原理是通过Object.defineProperty实现。

3. Compiler类负责解析指令和插值表达式（更新视图的方法）。

4. Dep类负责收集依赖、添加观察者模式。通知data对应的所有观察者Watcher来更新视图。在Observer类把每一个data转换成getter和setter时，会创建一个Dep实例，用来负责收集依赖并发送通知。在每一个data中在getter中收集依赖。在setter中通知依赖，既通知所有Watcher实例新视图。

5. Watcher类负责数据更新后，使关联视图重新渲染。

**new一个新对象的过程**

1. 先创建了一个新对象 var person={}；

2. 将这个对象的\_proto\_的指针指向了这个构造函数的原型对象prototype。

3. 将构造函数的作用域赋给这个新对象

4. 执行构造函数的内部代码，并将其属性和方法添加给person中的this对象。即Person.call(this). 继承构造函数的属性以及方法。

5. 返回这个新对象person

**this指向问题**

1. 在方法中 this表示的是方法所属的对象
2. 单独使用的时候this指向的是全局 即window
3. 在函数中this也是指向的全局对象 但是在严格模式的情况下 this是undefined
4. 在事件中，this指向的是接收事件的元素
5. call（）和apply（）方法可以使this指向任何一个元素对象

6 构造函数中的this 指向new出来的实例化对象

**Vue 不能检测以下数组的变动**

当你利用索引直接设置一个数组项时，例如：vm.items[indexOfItem] = newValue

当你修改数组的长度时，例如：vm.items.length = newLength

避免方案

1. 方法变更 Vue将监听能够改变数组本身的七个方法进行了包裹,从而达到触发视图更新的目的.

push() pop() shift() unshift() splice() sort()reverse()

1. 替换数组

filter()、concat() 和 slice()它们不会变更原始数组，而总是返回一个新数组。当使用非变更方法时，可以用新数组替换旧数组

1. 对象的更新检测

Vue无法检测property的添加和移除.由于Vue会在初始化实例时对property执行getter/setter转化, 所以property必须在data对象上存在才能让Vue将他转为响应式

**Vue和React的区别**

相同点:

1. 数据驱动视图

2. 组件化

都使用 Virtual DOM

不同点:

### **1. 核心思想不同**

Vue灵活易用的渐进式框架，进行数据拦截/代理，它对侦测数据的变化更敏感、更精确。本质是MVVM框架，由MVC发展而来。

React推崇函数式编程（纯组件），数据不可变以及单向数据流,是前端组件化框架，由后端组件化发展而来。

### **2. 组件写法差异**

React推荐的做法是JSX + inline style, 也就是把 HTML 和 CSS 全都写进 JavaScript 中,即 all in js;

Vue 推荐的做法是 template 的单文件组件格式(简单易懂，从传统前端转过来易于理解),即 html,css,JS 写在同一个文件(vue也支持JSX写法)

### **3. diff算法不同**

流程思路类似 :

不同的组件产生不同的 DOM 结构。当type不相同时，对应DOM操作就是直接销毁老的DOM，创建新的DOM。

同一层次的一组子节点，可以通过唯一的 key 区分。

react首先对新集合进行遍历，for( name in nextChildren)。

通过唯一key来判断老集合中是否存在相同的节点。如果没有的话创建

如果有的话，if (preChild === nextChild )

3.1会将节点在新集合中的位置和在老集合中lastIndex进行比较

3.2如果if (child.\_mountIndex < lastIndex) 进行移动操作，否则不进行移动操作。

如果遍历的过程中，发现在新集合中没有，但在老集合中有的节点，会进行删除操作

Vue2的核心Diff算法采用了双端比较的算法，同时从新旧children的两端开始进行比较，借助key值找到可复用的节点，再进行相关操作。相比React的Diff算法，同样情况下可以减少移动节点次数，减少不必要的性能损耗，更加的优雅。

### **4. 响应式原理不同**

Vue

Vue依赖收集，自动优化，数据可变。

Vue递归监听data的所有属性,直接修改。

当数据改变时，自动找到引用组件重新渲染。

React

React基于状态机，手动优化，数据不可变，需要setState驱动新的state替换老的state。当数据改变时，以组件为根目录，默认全部重新渲染, 所以Reac中会需要 shouldComponentUpdate 这个生命周期函数方法来进行控制

**原型继承**

//父类

function People(name,age){

this.name = name || 'wangxiao'

this.age = age || 27

}

//父类方法

People.prototype.eat = function(){

return this.name + this.age + 'eat sleep'

}

//子类

function Woman(name,age){

//继承父类属性

People.call(this,name,age)

}

//继承父类方法

(function(){

// 创建空类

let Super = function(){};

Super.prototype = People.prototype;

//父类的实例作为子类的原型

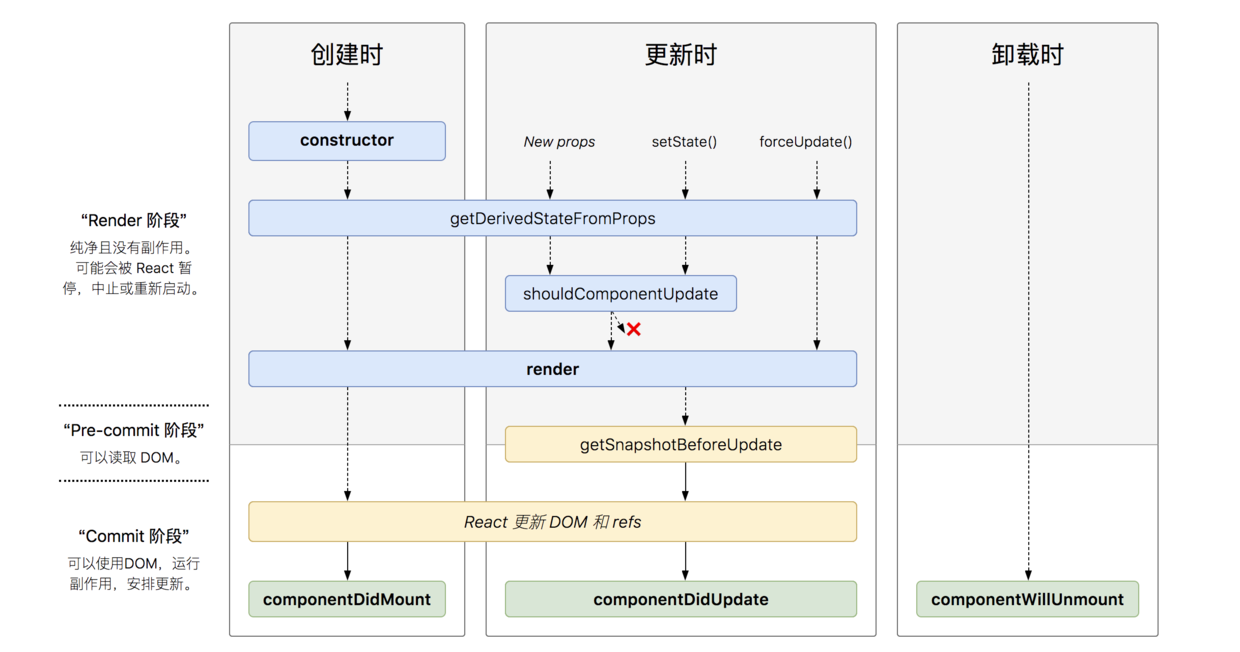
Woman.prototype = new Super();

})();

//修复构造函数指向问题

Woman.prototype.constructor = Woman;

let womanObj = new Woman();



**Vue 缺点**

1. 由于底层基于Object.defineProperty实现响应式，而这个api本身不支持IE8及以下浏览器  
   2、csr的先天不足，首屏性能问题（白屏）  
   3、由于百度等搜索引擎爬虫无法爬取js中的内容，故spa先天就对seo优化心有余力不足

**BFC**

BFC 全称为 块格式化上下文 (Block Formatting Context) 。

一个BFC区域只包含其子元素，不包括其子元素的子元素.

不同的BFC区域之间是相互独立的，互不影响的。利用这个特性我们可以让不同BFC区域之间的布局不产生影响.

产生条件

1. float 的值不是 none
2. position 的值不是absoulte或者fixed
3. display的值是inline-block、table-cell、flex
4. overflow 的值hidden，auto，scroll

解决问题

1. 解决外边距的塌陷问题（垂直塌陷）
2. 利用BFC解决包含塌陷
3. 当浮动产生影响的时候，可以利用BFC来清除浮动的影响
4. BFC可以阻止标准流元素被浮动元素覆盖

**keep-alive**

<keep-alive>包裹动态组件时，会缓存不活动的组件实例，而不是销毁它们。 和<transition>相似，<keep-alive>是一个抽象组件：它自身不会渲染一个 DOM 元素，也不会出现在组件的父组件链中。 当组件在 <keep-alive> 内被切换，它的 activated 和 deactivated 这两个生命周期钩子函数将会被对应执行。 他可以避免组件反复创建和渲染,有效提升系统性能,

在keep-alive中的在跳转到指定的路由时刷新对应的路由，其余不刷新。

1. 动态修改include先移除include 然后在组件的this.$nextTick中添加回去
2. 监听$router的变化
3. 使用activated和 deactivated这两个钩子
4. 使用v-if进行删除
5. 不同的路由使用的相同的组件,页面切换时不会触发created和mounted钩子, 需要手动监听$router 在router-view上添加唯一的key

**组件传值方式**

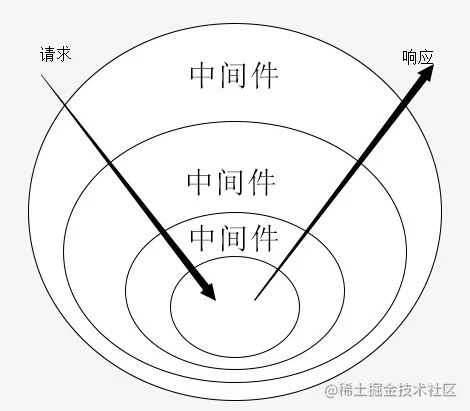
1. 父组件向子组件传值: props slot(插槽)
2. 子组件向父组件传值 子组件 $emit() 父组件 $on()
3. 父子之间调用: $refs $root $parent $children
4. 子子之间传值 放到父组件中 事件总线
5. 跨层级传值 依赖注入 provide/inject 使用Vuex进行状态管理

**css 模块化**

手写源生 CSS  
 使用预处理器 Sass/Less  
 使用后处理器 PostCSS  
 使用 css modules  
 使用 css in js   
 debugx5.qq.com

**洋葱模型**

koa的洋葱模型是以next()函数为分割点，先由外到内执行Request的逻辑，然后再由内到外执行Response的逻辑，这里的request的逻辑，我们可以理解为是next之前的内容，response的逻辑是next函数之后的内容，也可以说每一个中间件都有两次处理时机。洋葱模型的核心原理主要是借助compose方法。为了大家更好的理解什么是洋葱模型，



use方法

当我们使用中间件的时候，首先是使用use方法，use方法会将传入的中间件回调函数存储到middleware中间件数组中。

listen方法

当执行app.listen去监听端口的时候，其实其内部调用了http模块的createServer方法，然后传入内置的callback方法，这个callback方法就会将use方法存储的middleware中间件数组传给compose函数。

compose方法

compose方法是洋葱模型的核心，compose方法中有一个dispatch方法，第一次调用的时候，执行的是第一个中间件函数，中间件函数执行的时候就是再次调用dispatch函数，也就说形成了一个递归，这就是next函数执行的时候会执行下一个中间件的原因，因此形成了一个洋葱模型。

**圣杯布局 双飞翼**

圣杯布局

圣杯布局是两边固定宽度，中间自适应的三栏布局，其中，中间栏放到文档流前面，保证先行渲染。解决方案大体都是三栏全部float:left浮动，而在解决中间栏div的内容不被遮挡上，圣杯布局是中间栏在添加相对定位，并配合left和right属性，效果上表现为三栏是单独分开的。

双飞翼布局定

双飞翼布局是在中间栏的div中嵌套一个div，内容写在嵌套的div里，然后对嵌套的div设置margin-left和margin-right，

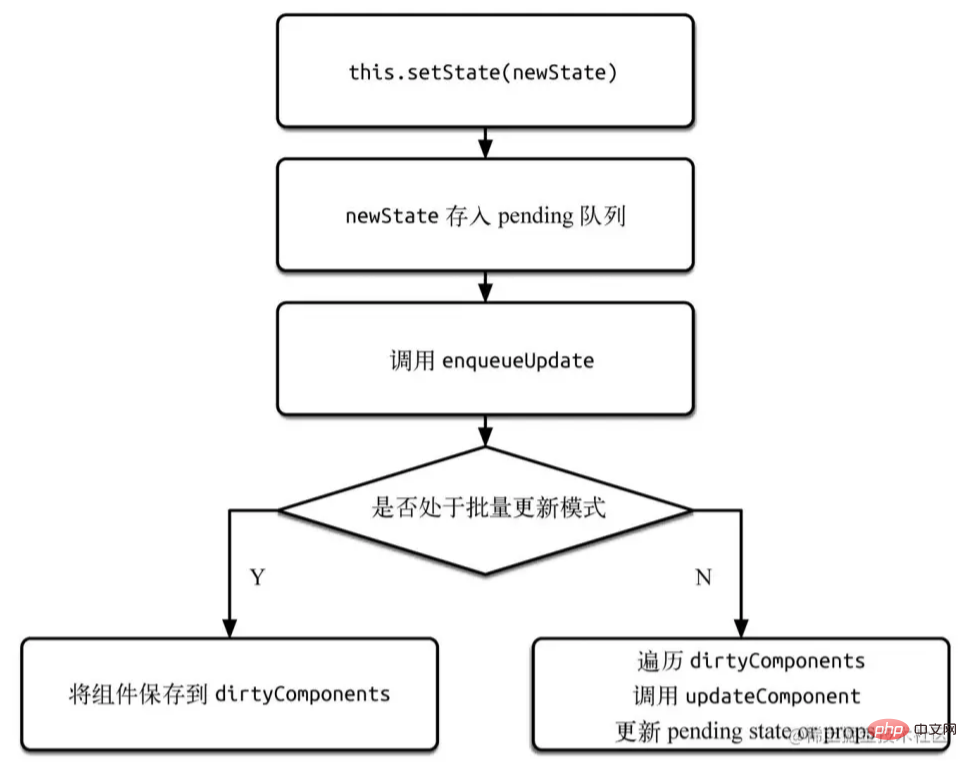
效果上表现为左右两栏在中间栏的上面，中间栏还是100%宽度，只不过中间栏的内容通过margin的值显示在中间。

**setState更新机制**

当调用setState时，实际上会执行 enqueueSetState 方法，并对 partialState 以及 \_pendingStateQueue 更新队列进行合并操作，最终操作 enqueueSetState 执行 state 更新。

而 performUpdateIfNecessary 方法会获取 \_pendingElement、\_pendingStateQueue、\_pendingForceUpdate，并调用 receiveComponent 和 updateComponent 方法进行组件更新。

如在 shouldComponetUpdate 或 componentWillUpdate 方法中调用 setState， 此时 this.\_pendingStateQueue != null, 则 performUpateIfNecessary 方法就会调用 updateComponent 方法进行组件更新，但 updateComponent 方法又会调用 shouldComponentUpdate 和 componentWillUpdate 方法，因此造成循环调用，使得浏览器内存占满后崩溃。



什么是babel?

babel是转码的工具、解析器（vue/jsx）

1. babel核心包及用处
2. @babel/cli：提供命令
3. @babel/core：核心库
4. @babel/preset-env：预设，转码es6—>es5（只转es5的核心包）
5. @babel/polyfill 垫片：可以把高端代码转译成低端浏览器所使用的； 可以兼容 IE8 IE7
6. @babel/runtime：运行时项目体积过大；或者说它代码里有些东西虽然实现了它但是它没有优化
7. @babel/plugin-transform-runtime：优化class

**Symbol set map**

Symbol 是 ES6 引入了一种新的原始数据类型，表示独一无二的值。

let s1 = Symbol('a');

let s2 = Symbol('a');

s1 === s2 //false

Symbol 值不能与其他类型的值进行运算，但可以转为布尔值，但是不能转为数值；

用于对象的属性名，可以保证不会出现同名的属性，对于一个对象由多个模块构成的情况非常有用，能防止某一个键被不小心改写或覆盖；

Symbol 作为属性名，不会被常规方法遍历得到，即该属性不会出现在for...in、for...of循环中，也不会被Object.keys()、Object.getOwnPropertyNames()、JSON.stringify()返回，但是，它并不是私有属性，可以使用 Object.getOwnPropertySymbols 方法，可以获取指定对象的所有 Symbol 属性名；

Set。它类似于数组，但是成员的值都是唯一的，没有重复的值，它 本身是一个构造函数，用来生成 Set 数据结构。

Set的遍历顺序就是插入顺序。

add(value)：添加某个值，返回Set结构本身。  
delete(value)：删除某个值，返回一个布尔值，表示删除是否成功。  
has(value)：返回一个布尔值，表示该值是否为Set的成员。

clear()：清除所有成员，没有返回值。

Map: 它类似于对象，也是键值对的集合，但是“键”的范围不限于字符串，各种类型的值（包括对象）都可以当作键，是一种更完善的 Hash 结构实现。遍历顺序就是插入顺序。

set(key，val)：添加某个值，返回Map结构本身。  
get(key)：     读取某个键，如果该键未知，则返回undefined  
delete(key)：  删除某个键，返回一个布尔值，表示删除是否成功。  
has(key)：     返回一个布尔值，表示该值是否为Map的键。  
clear() :      清除所有成员，没有返回值。

**webpack热更新**

启动webpack，生成compiler实例。compiler上有很多方法，比如可以启动 webpack 所有编译工作，以及监听本地文件的变化。

使用express框架启动本地server，让浏览器可以请求本地的静态资源。

本地server启动之后，再去启动websocket服务，如果不了解websocket，建议简单了解一下[websocket速成](https://www.ruanyifeng.com/blog/2017/05/websocket.html)。通过websocket，可以建立本地服务和浏览器的双向通信。这样就可以实现当本地文件发生变化，立马告知浏览器可以热更新代码啦！

**设计摸模式**

1、工厂模式

【简单的工厂模】：可以理解为解决多个相似的问题【提示框，只是提示的文字需要修改】

【复杂的工厂模式】：将其成员对象的实列化推迟到子类中，子类可以重写父类接口方法以便创建的时候指定自己的对象类型【各种UI组件，根据你要的类型不同（比如：按钮，提示框，表格等）】

2、 单例模式

两个特点：一个类只有一个实例，并且提供可全局访问点 全局对象是最简单的单例模式：window

demo:登录弹出框只需要实例化一次，就可以反复用了

3、模块模式

模块模式的思路是为单体模式添加私有变量和私有方法能够减少全局变量的使用  
demo:返回对象的匿名函数。在这个匿名函数内部，先定义了私有变量和函数

4、代理模式

代理对象可以代替本体被实例化，并使其可以被远程访问  
demo: 虚拟代理实现图片的预加载

5、缓存代理

缓存代理的含义就是对第一次运行时候进行缓存，当再一次运行相同的时候，直接从缓存里面取，这样做的好处是避免重复一次运算功能，如果运算非常复杂的话，对性能很耗费，那么使用缓存对象可以提高性能;

demo:计算值的加法，如果之前已经算过，取缓存，如果没有算过重新计算。

6、命令模式

有时候需要向某些对象发送请求，但是并不知道请求的接收者是谁，也不知道请求的操作是什么，此时希望用一种松耦合的方式来设计程序代码;使得请求发送者和请求接受者消除彼此代码中的耦合关系。

demo：几个按钮绑定不同的事件，然后bindEvent(el, event);

7、模板方法模式

一、模板方法模式：一种只需使用继承就可以实现的非常简单的模式。

二、模板方法模式由两部分组成，第一部分是抽象父类，第二部分是具体的实现子类。

demo: 比如泡茶、冲咖啡的步骤都是一样的，抽出父类，Child.prototype = new Parent();然后重写里面的步骤（方法）

8、策略模式

定义一系列的算法，把它们一个个封装起来，并且使它们可以相互替换

demo：年终奖的薪水的几倍，是按照一个个等级来划分的，A级别是3倍，B级别是2倍，C级别是1倍，那么就可以写三个等级方法，然后封装在一个方法里，传入薪水和等级就ok了

9、发布订阅模式介绍

发布—订阅模式又叫观察者模式，它定义了对象间的一种一对多的关系，让多个观察者对象同时监听某一个主题对象，当一个对象发生改变时，所有依赖于它的对象都将得到通知。

demo: 比如你向买房，只要把手机给房产中介，房产中介一有消息就发布消息。

10、中介者模式

中介者模式的作用是解除对象与对象之间的耦合关系，增加一个中介对象后，所有的相关对象都通过中介者对象来通信，而不是相互引用，所以当一个对象发送改变时，只需要通知中介者对象即可。中介者使各个对象之间耦合松散，而且可以独立地改变它们之间的交互。

demo:卖手机，颜色和数量判断加入购物车按钮是否可用

11、装饰者模式

动态的给类或对象增加职责的设计模式。

装饰器模式并不去深入依赖于对象是如何创建的，而是专注于扩展它们的功能这一问题上。装饰器模式相比生成子类更为灵活。

12、适配器模式

适配器模式主要解决两个接口之间不匹配的问题，不会改变原有的接口，而是由一个对象对另一个对象的包装。  
demo:两个地图（2个类），他们有一个共同方法但是名字不同，这时候需要定义适配器类, 对其中的一个类进行封装。