# 前端知识点

# 一、js基础

## 1、执行上下文/作用域链/闭包

（1）简单来说，执行上下文是js代码运行时所在的环境的抽象，主要定义变量或函数等数据，以及他们的操作。分为全局和局部（函数）执行上下文。每个执行上下文都有一个与之关联的变量对象，而环境中定义的所有变量和函数都保存在这个对象中。

而当代码在一个环境中执行时，会创建变量对象的一个作用域链，而这个作用域链能保证变量和函数的有序访问。

（2）闭包：是指有权访问另一个函数作用域中的变量的函数。

两点：内部函数，具有外部变量的引用。

应用场景：定时器，函数防抖



## 2、this/call/apply/bind

（1）this：函数内部的属性，this引用的是函数执行时的执行上下文对象

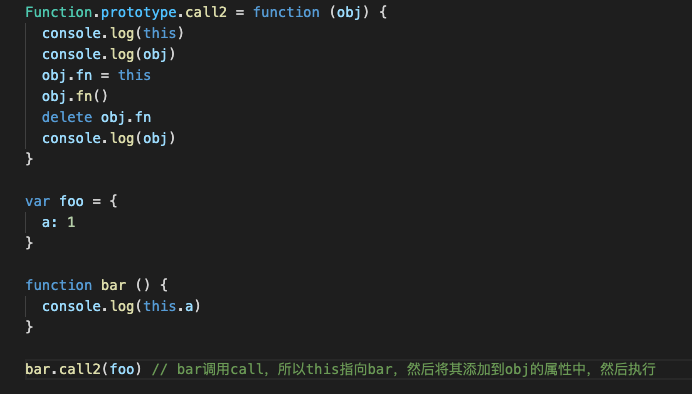
（2）改变this的指向：call()、apply()、bind()和new一个对象

（3）call和apply的区别：call和apply都是将this重新指向第一个参数，不同的是，apply只能接收两个参数，第二参数是一个数组，而call接收多个参数。严格模式下，this值就是call和apply的第一个参数，无论是null还是undefined。非严格模式下，如果this被指定为null或者undefined，那么会自动重新指向window。

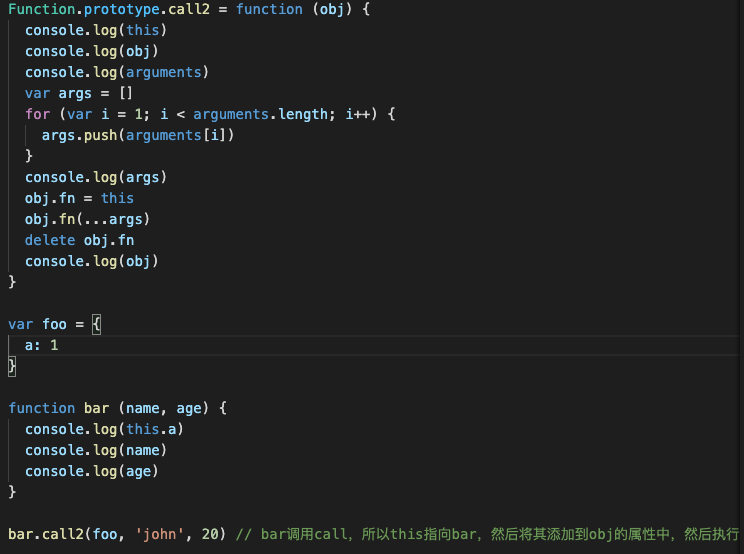
（4）call和apply的模拟实现：

想法：当我们调用fn,call(0bj)时，实际上可以把fn变成obj的一个属性，这样再执行这个函数，执行完删除这个属性，就完成了目标。

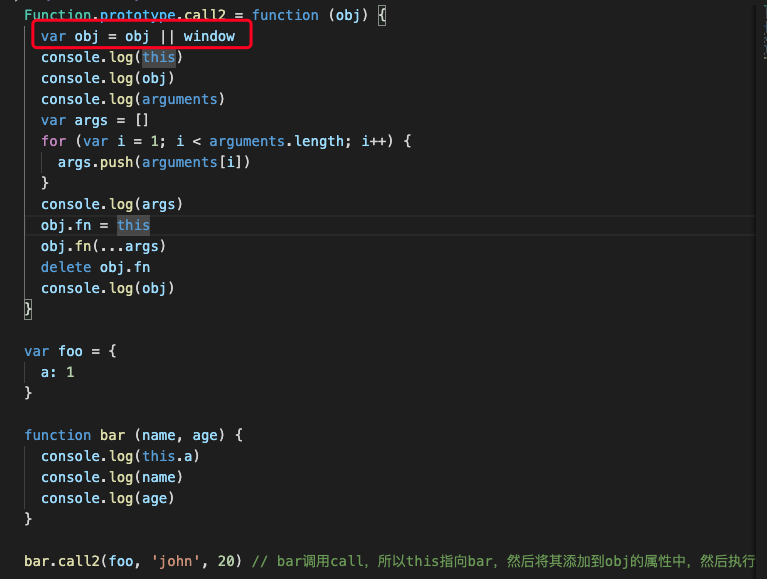
那么其简易实现如下：



接着，call2里需要可以指定参数，进一步改进，也就是将参数都封装到一个数组里，然后解构就行了



虽然解决了参数问题，但是还有当传入的第一个参数为null或者undefined时，this指向window。这个实现就相对简单些。只需要将obj重新赋值下即可



这样就完成了一个简易的call2函数，apply的实现与之类似。

（5）bind函数：也是改变this的指向，指向第一个参数，和call和apply不同的是，bind绑定不会立即执行函数。如果要执行，那么就需要加一个括号。并且bind只生效一次，永久绑定到第一个对象上。

## **3、原型/继承**

（1）原型和原型链

简单来说，我们没创建一个函数，它都有一个prototype属性，这个属性就是原型，通常用来保存公共属性和方法。其原型里包含一个constructor指针，其指向函数本身。而函数的实例也包含一个\_\_proto\_\_属性，指向函数的原型。

类似java中，所有类都是Object的子类，js中，所有函数对象中的原型中都有一个\_\_proto\_\_属性指向Object，也就是说，函数的原型是Object实例。这其中就形成了一个原型链。

（2）如何利用原型实现继承

继承就是让子类拥有父类的所有属性和方法。那么要用原型实现继承，只需要将父类的所有属性和方法都绑定到子类的原型上，那么子类就获得了父类的所有属性和方法，前面说过，函数的实例对象包含一个指向父类原型的\_\_proto\_\_指针，而实例对象包含构造的所有属性，那么我们只需要将父类的实例对象赋值给子类的原型，那么子类的原型就包含了父类实例对象上的属性，也就是包含了\_\_proto\_\_和父类的属性。从而实现了继承。

（3）实现继承的几种方式：

1、原型链，也就是上面红色部分所说的

问题：当父类中的属性是一个引用类型时，在子类中修改属性，会导致所有其他子类的属性也跟着改变。 同时，这种方式创建子类的实例时，不能项父类型的构造函数中传递参数。

2、借用构造函数

也就是在子类型的构造函数中使用call(this))或者apply(this)方法。

SuperType.call(this)，就把父类中的this指向了之类，那么父类中的this.color等属性就属于子类型了，也就实现了继承。这种方式可以向父类构造函数中传参。

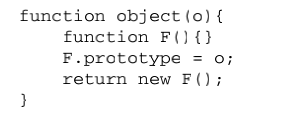
问题：也很明显，所有属性和方法都要写在父类型的构造函数上，不能在原型中，否则引用不到。

3、组合继承

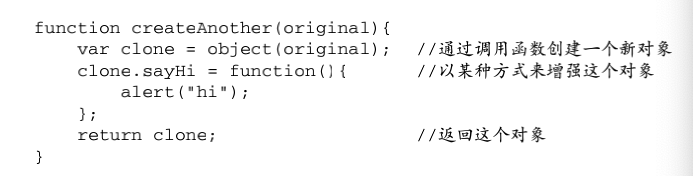
也就是将原型链继承和构造函数组合起来使用，也就是既在子类构造函数中，通过父类型构造函数call(this)，也将父类型的实例赋值给子类型的原型。

4、原型式继承

声明一个函数，传递一个对象，在函数内部新声明一个函数，令这个的原型等于传递过来的对象，最后返回这个函数的一个实例，返回的实例也就有了这个对象的所有属性和方法。



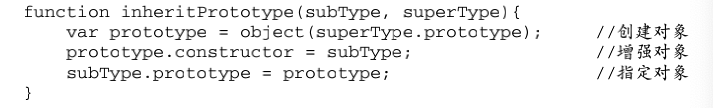
5、寄生式继承。



6、寄生组合式继承

组合式继承方式需要调用两次构造函数，分别是父类构造函数call(this)时和new 一个父类型实例时。

寄生组合式继承的想法是，将第二次调用的构造函数替换为寄生式的方式。不去调用构造函数，而是通过object(superType.prototype)赋值一份父类型的原型，并赋值给子类型的原型。



个人觉得还不如直接调用构造函数，虽然重复了一次实例属性的赋值。

## **4、Promise**

（1）Promise是什么

Promise是实现异步编程的一种新的解决方案，相比于旧的解决方案有两点优点：

1、不必在每次执行异步操作时，事先指定回调函数。

2、可以解决回调地狱问题。

Promise有几个方法，原型的上的方法有.then和.catch Promise的构造函数方法有resolve, reject,all,race等

（2）Promise的手写实现：

<https://github.com/WangCheng0928/promise>

（3）宏任务和微任务

这就涉及到事件循环了！

宏任务：I/O操作，setTimeout,setInterval,ajax,dom事件回调

微任务：MutationObserver(观察dom节点的变化)，Promise.

由于js是单线程的，所以一般非阻塞的同步任务会放在主线程中执行，而异步任务就会由事件管理模块管理，放在回调队列中，等待同步任务执行完后，再去执行异步任务。那么对于宏任务和微任务，它们会分别放到宏队列和微队列中。那么就有这么一个规定，在准备执行宏任务之前，必须先把微任务一个个执行完。

由上可以总结出：js执行代码时，首先在主线程中执行同步代码，对于异步任务，如果是宏任务会将其放到宏队列中，如果是微任务，就将其放到微队列中，待同步任务执行完后，在执行每一个宏任务执行之前，首先得把微队列里的微任务一个一个执行完。然后再去执行宏任务。如果宏任务里又有微任务，那么微任务再次会放到微队列里。那么会在执行下一个宏任务之前，首先把微队列里任务执行完，再去执行宏任务。

（4）async await

async和await是目前解决异步编程的终极解决方案

使用async修饰的函数最终会返回一个Promise对象，然后可以用.then方法得到里面的结果，或者使用await 可以得到异步操作的结果。其原理是将generator和自动执行器封装在一个函数里。

## **5、深浅拷贝**

（1）基本概念

深浅拷贝都是对于js中的引用类型而言的，浅拷贝就指示复制对象的引用（地址），如果拷贝后对象发生变化，原对象也会发生变化。只有深拷贝才是真正的对对象的拷贝。

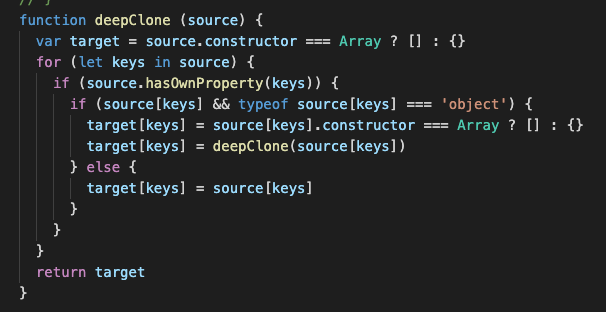
浅拷贝：object.assign()方法， = 赋值

深拷贝：1、利用json对象中的parse和stringfy

2、利用递归实现每一层都重新创建对象并复制。

其中，利用json.parse(json.stringfy())只能适用一些简单的对象，而对于对象中包含undefined、function和symbol就会导致这些属性丢失。

深拷贝的递归实现：

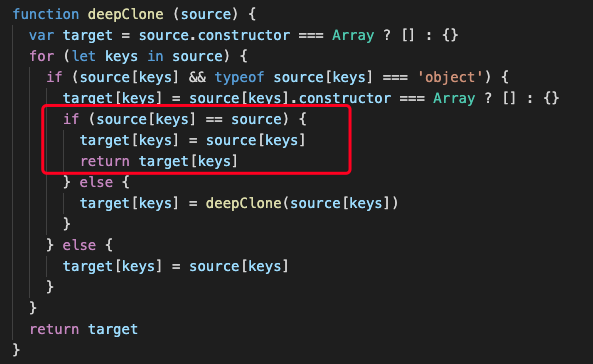


数组的concat，slice，object的assign方法都是只对对象的第一层进行深拷贝。包括...展开符也是一样

（2）循环引用问题

当要拷贝的对象的一个属性是对象本身时，例如obj.z=obj时，这时候就会出现循环引用问题。导致报错。

解决办法：只需要在赋值的时候判断下source[keys]=source即可



## **6、事件机制/Event Loop**

（1）如何实现一个事件的发布订阅



最简易的实现方式，运用到队列，首先设置一个监听器，表示要监听的事件名称，以及当监听到后，怎么处理。然后发布事件的时候只需要，传事件名和参数，就能自动触发事件的执行。

（2）事件循环/Event Loop

在前面已经将过了，不过这里需要注意浏览器的执行机制。

当我们在浏览器中输入网址后，会发生哪些事情？

1、DNS查询：查询顺序：浏览器dns缓存、本地dns缓存、本地hosts缓存、路由器dns、isp的dns服务器、根服务器。查询到后就可以直接访问ip地址了。

2、建立连接：tcp/ip协议，三次握手和四次挥手

3、发送请求

4、服务器响应数据

5、浏览器接受到html页面：这里才真正到了浏览器该做事的时候了。

此时的具体步骤是：1、解析html文本，构建dom树，2、构建呈现树RenderTree，这一步主要是将css样式附着在DOM节点上。3、布局、也就是节点应该呈现在什么地方

4、绘制，将呈现树绘制出来，显示在屏幕上。

需要注意的是html文本自上而下解析，遇到任何样式和脚本都会阻塞。

1、css加载不会阻塞html文件解析，但会阻塞dom的渲染

2、css加载会阻塞后面js语句的执行

3、js会阻塞html的解析和渲染

回到event loop中，渲染更新会在event loop中的任务宏任务和微任务完成后执行，但并不是每轮event loop都会更新渲染，这取决于是否修改了dom和浏览器是否觉得有必要在此时立即更新状态呈现给用户。如果在一帧的事件内（16.7ms）修改了多处dom，浏览器可能将变动积攒起来，只进行一次绘制。如果希望在每轮event loop都即时呈现，可以使用requestAnimationFrame.

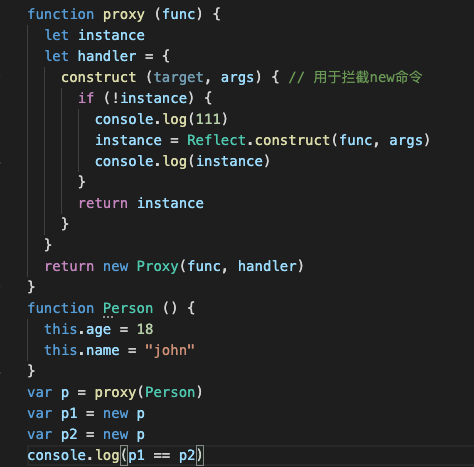
## **7、web worker**

web worker 是html5标准的一部分，包含一套api，它允许js程序运行在主线程之外的另一个线程中。通过new worker(xxx.js)创建一个实例对象，通过worker.postMessage(发送数据)，主线程就可以通过woker.onMessage来接受到数据。

## 附：

### 单例模式：

proxy，相当于一个拦截层，在执行函数前



### 柯里化：

将传多个参数的函数转换成传一个参数的函数，返回一个传剩余参数的函数。

柯里化的作用：延迟运行、参数复用。

### js里面的异步操作：

1、回调函数

2、事件监听

3、观察者模式

4、promise

5、async await

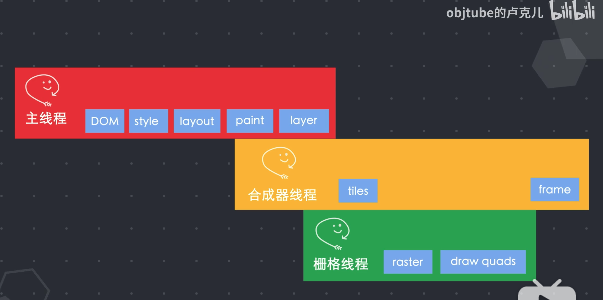
6、co库的generator函数

### 为什么要避免大量的重绘会重排？

浏览器进程负责除标签外的用户界面（地址栏、书签、后退按钮）以及负责与浏览器其他进程协调工作。网络进程负责发起网络请求，GPU进程负责整个浏览器的渲染。渲染器进程用来控制tab标签内的所有内容。

浏览器的网络进程在获得请求的html数据后，通过IPC将数据传给渲染器进程的主线程。主线程将html文本进行解析，构建dom树，并进行样式计算（renderTree，附着）。接着根据renderTree进行布局，生成layoutTree，通过遍历layoutTree生成绘制顺序表。接着遍历layeroutTree生成layerTree，然后主线程将layerTree和绘制顺序表传给合成器线程，合成器线程按规则进行分图层，并把图层分为更小的图块传给栅格线程进行栅格化，栅格化完成后，合成器线程会收到栅格化传过来的draw quads图块信息，接着合成器线程将这些图块信息合成合成器帧，然后通过IPC传给浏览器进程，浏览器接着传给gpu进程，gpu渲染合成器帧，从而呈现给用户。

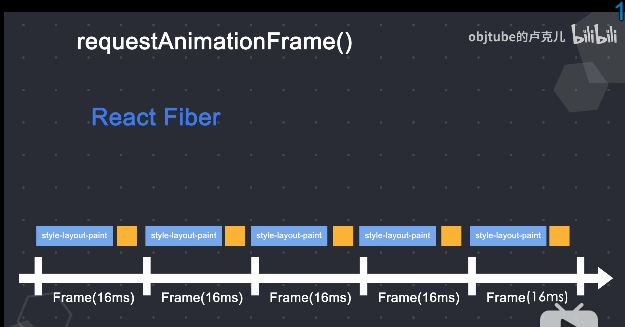




反复进程重绘或者重排会导致掉帧，浏览器卡顿。这是因为，在浏览器主线程绘制完一帧后，如果空闲出来，就会执行相应的js代码，如果js代码执行时间较长，那么绘制下一帧的时间时间就比较长，此时人眼便可以分辨出来卡顿现象。通过requestAnimationFrame() api可以解决这个问题，它在每一帧绘制前先暂停前面的js执行，然后调用传入的callback函数进行渲染，从而保证下一帧的顺利渲染。

另一种方法是通过transform动画，可以避免浏览器卡顿，这是因为transform

不会导致重绘或重排，而是直接运行在合成器线程和栅格线程中，从而不会阻塞主线程。

****

### 9种前端跨域解决方案

1、jsonp，其原理就是利用script标签无跨域限制，发请求给后台，同时将处理函数传给后台，后台再把数据和传过去的函数返回，浏览器就会执行该函数，从而完成跨域请求。

问题：JSONP只能处理get请求。安全性也不好。

2、CORS跨域资源共享

客户端正常发送请求，而在服务端设置允许请求的header。在header添加客户端的请求地址，请求方式等。这是一种比较常见的方式。

问题：支持多源时，写\*号，但是如果写\*号，那么就不能使用cookie了。如果硬想支持多源，那么就是多写几个。

3、http proxy代理 （常用的方式）

需要在webpack中配置proxy

4、nginx：反向代理

5、postMessage

window.postPostmessage, window.onmessage

6、socket.io(websocket，客户端与服务端实时通信的协议)

io(https://www.xxx.com)与服务器建立连接

7、document.domain + iframe

只能实现同一个主域，不同子域之间的操作。

8、window.name + iframe

9、location.hash+ iframe

### 网络安全

XSS攻击

Cross-Site Scripting(跨站脚本攻击)简称XSS,是一种代码注入攻击，攻击者通过在目标网站上注入恶意脚本，使之在浏览器上运行。利用这些脚本，攻击者可获取用户的敏感信息如cookie、sessionid等，进而危害数据安全。

xss的本质是：恶意代码未经过滤，与网站正常代码混在一起；浏览器无法分辨哪些脚本是可信的，导致恶意脚本被执行。

类型：存储型、反射型和dom型

预防：输入过滤（escapeHTML）、纯前端渲染、html转义等。

CSRF攻击

Cross-site request forgery 跨站请求伪造：攻击者诱导受害者进入第三方网站，在第三方网站中，向攻击者网站发送跨站请求。利用受害者在被攻击网站已经获取的注册凭证，绕过后台的用户验证，达到冒充用户对被攻击的网站执行某项操作的目的

防护策略：

同源检测

CSRF token ，这个token都放在服务器的session中。加载页面时，首先从服务器拿到token，发送请求时，首先验证这个token。

双重cookie验证，前端发起请求时，取出cookie添加到url的参数中，后台取出参数，验证与cookie中的字段是否一致，不一致则拒绝。

### session & cookie

1、为什么需要session或者cookie呢？

我们用户通常都是在应用层进行操作。应用层写http是无状态的，也就是在用户发送请求，服务器返回数据后，连接就断开了，再次交互就要建立新的连接。因此，服务器无法通过连接跟踪会话。那怎么办呢？就需要一种缓存机制，来存储会话信息。

浏览器采用的方式就是cookie，服务器端采用的方式就是session。

cookie：如果服务端想要记住用户的状态，就使用response想客户端发送一个cookie。客户端会把cookie保存起来。再次发送请求就会将cookie发送给服务器，这样服务器就能从cookie读取信息，知道是该用户发过来的请求了。

session：session是服务器的缓存机制，当服务器与客户端建立连接后，服务器会自动分配一个sessionId。每次请求带着这个sessionID，那么每次请求，服务器可以在这个sessionId对应的session中拿到用户信息。

2、两者的区别

从上面可以看出，两者明显的区别是：

1、cookie存储在客户端，session存储在服务端。

2、既然cookie存在客服端，那么就有被劫持的危险，因此其安全性较低。

3、session存在服务器，那么当session过多的话，就会给服务器造成压力。

4、cookie的大小有限制，单个cookie保存的数据不能超过4k。

5、当会话结束关闭浏览器，session就会销毁。

# css基础

## 1、position

relative和absolute的区别

relative，顾名思义，相对定位，元素没有脱离文档流，相对于原来文档流中的位置进行定位。

absolute：绝对定位，脱离了文档流，相对于static定位以外的第一个父元素进行定位

fixed：相对于浏览器窗口进行定位

static: 没有定位，元素出现在正常的流中。

inherit：从父元素继承position属性的值

## **2、行内元素/块状元素**

行内元素：可以与其他行内元素共享一行，不会独占一行。

不能更改元素的height、width的值，大小由内容撑开

margin只有left和right产生边距效果，top和bottom就不行

块级元素：独占一行，在不设置自己的宽度的情况下，默认填满父级元素的宽度。

能够设置height、width的值

可以使用padding，margin的各个属性值。

## **3、flex布局**

（1）介绍下flex布局

flexible box的缩写

（2）flex:1指的是什么

flex:1表示 flex: 1 1 0%,也就是flex-grow 1 flex-shrink 1 basis 0%

（3）flex-shrink 和 flex-basis属性

flex-shrink默认为1，即如果空间不足，将该项目缩小。

flex-basis属性定义了在分配多余空间前，项目占据的主轴空间。默认auto，根据项目本身计算。比如字数多，就大一些。

如果当前元素超出了容器，则会根据flex-basis属性来计算缩放的尺寸。

## **4、水平居中的方法**

（1）元素为行内元素，设置父元素text-alig: center

（2）如果元素宽度固定，可以设置margin:auto

（3）如果元素宽度固定，通过使用绝对定位，以及设置元素margin-left为其宽度的一半

（4）使用flex布局，justify-content属性center

## **5、css3新特性**

1、新增flex布局

2、选择器新增nth-child等选择器

3、盒模型方面添加了box-sizing来改变盒模型

4、在动画方面新增了animation、2d、3d变换等。颜色方面添加透明、rgba

5、字体阴影和盒子的阴影

6、媒体查询

## **6、讲讲盒模型**

分为content box和border-box

区别border-box 盒模型的狂傲是content+padding+border的总宽高

## **7、css选择器分类**

不同级别：排序: !important > 行内样式 > ID选择器 > 类选择器=伪类=属性 > 标签=伪元素 > 通配符 > 继承 > 浏览器默认属性

## **8、清除浮动有哪些方法**

1、利用clear属性

在父元素离创建一个空元素，对其设置clear:both的样式

2、使用clear属性+伪元素

.xxx::after{content:’’;display:block;clear:both;visibility:hidden;height:0}

3、使用BFC布局规则

position为absolute或fixed

overflow不为visible的块元素

display为inline-block,table-cell,table-caption

## **9、BFC**

1、同一个BFC下外边距会发生折叠

2、BFC可以包含浮动的元素（清除浮动 overflow:hidden）

3、可以阻止元素被浮动元素覆盖

<https://zhuanlan.zhihu.com/p/25321647>

# vue基础

## 1、vue动态路由加载

（1）对于不同参数加载同一组件，可以使用路由参数来实现，格式在/path/:param

（2）匹配任意路径，使用通配符\*

（3）通过router.addRoutes( )api来动态添加路由。

## **2、webpack中的alias配置**

作用：设置别名是为了让后续引用的地方减少路径的复杂度

## **3、vue如何实现双向绑定**

vue数据双向绑定是通过数据劫持结合观察者模式的方式来实现的。

（1）数据劫持

当我们new Vue(el:, data:)时，我们需要拿到数据data，并将data通过Object.defineProperty()的方式重新定义，这就是数据劫持，通过设置访问器属性(configurable, enumerable, get, set)，这样才能在vue内部监听到数据的变化。

（2）发布订阅模式

监听器observer：对数据对象进行遍历，利用Object.defineProperty() 对数据进行劫持，从而监听数据的变化（注意Object.defineProperty()和proxy()方法的区别）

解析器compiler：也就是获取dom节点，解析vue模版指令，将模版中的变量都替换成数据，然后重新渲染视图页面。  
 订阅者Watcher：当数据发生变化时，触发解析器中的更新函数来更新view。

订阅器Dep：用来保存需要操作的watcher实例，当Observer到数据的变化时，Observer通知Dep数据变化，Dep接受到通知后，通过调用notify()方法来调用watcher的update()方法。

上述只是简单的原理，源码等具体的以后再深入理解分析。

## **4、vue生命周期**

beforeCreate: 此时刚初始化了一个空的Vue实例，el, data, methods未被初始化，那么此时可以用于显示loading...

created: 此时data和methods都已经初始化好了！因此，如果想要调用methods中的方法，最早可以在created中操作。

beforeMount: 此时模版已经在内存中初始化好了，但是还没有挂载到页面中去。el中的值也还没有进行赋值。

mounted：完成挂载，渲染值。如果想通过某些插件操作页面上的dom节点，最早是在mounted中进行。此时，组件已经脱离了创建阶段，进入到了运行阶段。

运行阶段只有两个生命周期函数beforeUpdate和updated.

beforeUpdate:此时监听到数据发生变化，但是view层没有被重新渲染，view层的数据还是旧的

updated:在updated执行之前，vue会根据最新的data，重新渲染出一份最新的内存dom树。内存dom树渲染完后，重新渲染到页面中去。那么，在updated执行的时候，页面和data已经保持同步了，都是最新的。

beforeDestroy:此时已经进入到实例的摧毁阶段，但是实例仍然完全可用

destroyed：钩子函数在vue实例销毁后执行。此时，vue实例指示的所有东西都会解绑定，所有的事件监听器都会被移除。所有的子实例都会被销毁。

## **5、vue的父子组件如何通信的**

父组件通过props属性给子组件通信

子组件通过触发事件来改变父组件的数据。

兄弟组件的通信，采用观察者模式，

## **6、Vue路由的实现：hash模式和history模式**

hash模式：在浏览器中符号“#”，#以及#后面的字符称之为hash，用window.location.hash读取。

特点：hash虽然在url中，但不被包含在http请求中，用来指导浏览器动作，对服务端安全无用，hash不会重加载页面。hash模式下，仅hash符号之前的内容会被包含在请求中，因此对于后端来说，即使没有做到路由的全覆盖，也不会返回404。

hash值的改变，都会在浏览器的访问历史中增加一个记录。因此，我们可以通过浏览器的回退、前进按钮控制hash的切换（实测，后退页面不更新，前进，页面重新跳转到相应url）。我们可以使用hashchange事件来监听hash值的变化，从而对页面进行跳转。

history模式：采用h5新特性，可以对浏览器历史记录栈进行修改。

histroy模式下，前端的url必须和实际向后端发起请求的url一致，如http://www.xxx.com/items/id。后端如果缺少对/items/id的路由处理，将返回404错误。

其中最主要的两个API:history.pushState()和history.replaceState(),这两个api也可以在不刷新页面的情况下，操作浏览器的历史记录。前者新增一个历史记录，后者直接替换当前的url。

我们可以使用popstate事件监听历史栈的变化，从而进行页面跳转。

## **7、对keep-alive的理解**

keep-alive是vue内置的一个组件，可以使被包含的组件保留状态，或避免重新渲染。

通过设置include和exclude属性来设置需要缓存的组件，那么之后跳转到相应的组件不会再次请求数据，直接使用缓存页面。

## **8、vue中key值的作用。**

当vue使用v-for更新已渲染过的列表时，默认使用“就地复用”策略，也就是如果更新的节点类型不同，那么用新的节点直接替换掉原来的节点，但是如果新节点类型和原节点类型相同，那么就会复用原来的节点，只是更改其属性。那么，如果插入某一相同类型的节点时，会直接复用原来的节点，只改属性，此节点的后续的节点也会重复操作。导致dom操作效率低。一句话：key的作用为了高效的更新虚拟dom。

## **9、vuex原理**

vuex是一个专为vue.js应用程序开发的状态管理模式。类似于react的redux，其核心概念包括State, Getter, Action, Mutation, Module。

（1）每一个vuex应用的核心就是store，存储着应用的大部分状态state。vuex的状态是响应式的。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更新。

（2）我们不能直接改变store中的状态。改变store中的状态的唯一途径就是显式的提交（commit）mutation。

state：单一状态树，在state中我们需要定义我们所需要管理的数组、对象、字符串等

gettters: 类似于vue中的计算属性，主要用来过滤一些数据。

mutation: 更改store中状态的唯一方式就是提交mutation，store.commit。

actions：用于提交mutation,不会直接变更状态。在action里可以进行异步操作，而mutation必须是同步操作。view层通过store.dispatch来分发action。

module：用来将store分模块管理，每个module都有自己的state, action 和getter。类似于redux中的reducer。

## **10、Proxy与Object.defineProperty的优劣对比？**

Proxy的优点：

proxy可以直接监听对象而非属性

proxy可以直接监听数据的变化

proxy返回的是一个新对象，那么我们就可以直接操作这个对象，而object.defineProperty只能遍历对象属性直接修改。

Object.defineProperty将会被废弃，都改成Reflect.defineProperty。

## **11、vue的父组件和子组件生命周期钩子函数执行顺序？**

加载渲染过程

父beforeCreate -> 父created->父beforeMount->子beforeMount->子created->子beforeMount->子mounted->父mounted

子组件更新过程

父beforeUpdate -> 子beforeUpdate->子updated->父updated

父组件更新过程

父beforeUpdate-> 父updated

摧毁过程

父beforeDestroy->子beforeDestroy->子destroyed->父destroyed

## **12、组件中的data为什么是一个函数？**

子组件通常是用来复用的，如果data是一个对象，那么其作用域没有隔离，子组件中的data属性值会相互影响。而如果是一个函数的话，那么每个实例都有一个对象的拷贝，不会相互影响。

## **13、使用过vue SSR吗？说说SSR?**

SSR服务端渲染，我的理解是，服务端将html文本和数据准备好，然后直接发送给浏览器，然后进行渲染。

优点：

更好的SEO

更快的内容到达时间（首屏加载更快）

## **14、说说你对SPA但页面的理解，它的优缺点是什么？**

SPA(single-page application) 仅在web页面初始化时加载相应的html、JavaScript、css。一旦页面加载完成，spa不会因为用户的操作而进行页面的重新加载或跳转；取而代之的是利用路由机制实现html内容的变换，ui与用户的交互，避免页面的重新加载。

优点：

1、用户体验好、快，内容的改变不需要重新加载整个页面，避免了不必要的跳转和重复渲染。

2、对服务器压力小。

3、前后端职责分离，架构清晰，前端进行交互逻辑，后端负责数据处理。

缺点：

1、初次加载耗时多。

2、前进后退由路由管理，不能使用浏览器的前进后退功能，所有页面切换需要自己建立堆栈管理

3、SEO难度大。所有的内容都在一个页面中动态替换展示。

## **15、对于即将到来的vue3.0特性有什么了解的吗？**

1、检测机制的改变，3.0采用基于proxy的observer实现。

2、组件生命方式，采用类式的写法，源码也采用typescript重写

3、vue3.0的插槽改变不会导致父组件重新渲染。提升了渲染性能。

# React基础

## 1、react有哪些组件？

（1）函数/无状态组件

函数或无状态组件是一个纯函数，它可接受参数，并返回react元素。这些组件没有状态和生命周期方法。

（2）类/有状态组件

类或有状态组件具有状态和生命周期方法，可以通过setState()方法来更改组件的状态。

（3）受控组件

受控组件是在react中处理输入表单的一种技术。表单元素通常维护它们自己的状态，正好可以将两者结合起来以控制输入表单。这称为受控组件。

（4）非受控组件

大多数情况下，建议使用受控组件。有一种称为非受控组件的方法可以通过使用Ref来处理表单数据。在非受控组件中，Ref用于直接从dom访问表单值。

简单来说，受控组件通过state获取状态数据，而非受控组件直接操作dom。

（5）容器组件

容器组件是处理获取数据、订阅redux存储等的组件。它们包含展示组件和其他容器组件，但是里面从来没有html。

（6）高阶组件

高阶组件是将组件作为参数并生成另一个组件的组件。

## 2、React生命周期

目前React16.8+的生命周期分为三个阶段，分别是挂载阶段，更新阶段和卸载阶段。

1、挂载阶段：

constructor：构造函数，最先被执行，我们通常在构造函数里初始化state对象或者给自定义方法绑定this。

getDerivedStateFromProps:static getDerivedStateFromProps(nextProps,prevState)这是个静态方法，当我们接受到新的属性想去修改我们的state，可以使用这个方法。

render：render函数是纯函数(输出只依赖于参数)，只返回需要渲染的东西，不应该包含其他的业务逻辑，可以返回原声的DOM、React组件、Fragment、Portals、字符串和数字、Boolean和null等内容。

componentDidMount: 组件装载之后调用，此时我们可以获取dom节点并操作，比如对canvas，svg的操作，服务器请求，订阅等。但是记得在componentwillUnmount中取消订阅。

2、更新阶段：

getDerivedStateFromProps：此方法在更新和挂载阶段都可能会调用

shouldComponentUpdate: shouldComponentUpdate(nextProps, nextState),有两个参数nextProps和nextState，表示新的属性和变化之后的state，返回一个布尔值。true表示会触发重新渲染，false表示不会触发重新渲染，默认返回true，我们通常利用此声明周期来优化React程序性能。

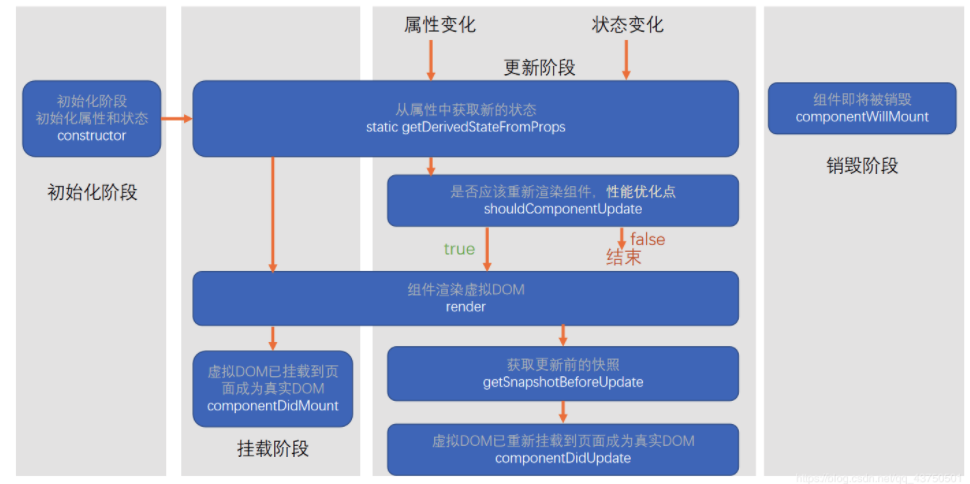
render：挂载阶段和更新阶段都有的函数，用来渲染虚拟dom。

getSnapshotBeforeUpdate: getSnapshotBeforeUpdate(prevProps,prevState),用来获取更新前的props和state快照，返回值会作为componentDidUpdate的第三个参数。

componentDidUpdate: componentDidUpdate(prevProps, prevState,snapshot) 第三个参数是getSnapshotBeforeUpdate返回的，如果触发某些回调函数时需要用到 Dom元素的状态，则将对比或计算的过程迁移至getSnapshotBeforeUpdate,然后在componentDidUpdate中统一触发回调或更新状态。

3、卸载阶段

componentWillUnmount:当我们组件被卸载或者销毁了就会调用，我们可以在这个函数里清除一些定时器，取消网络请求，清理无效的dom元素等垃圾清理工作。



## 3、setState到底是异步还是同步？

有时表现出异步，有时表现出同步。

1、setState通过一个队列机制来实现state更新，当执行setState()时，会将需要更新的state浅合并后放入状态队列，而不会立即更新state，队列机制可以高效的批量更新state。而如果不通过setState，直接修改this.state的值，则不会放入状态队列，当下一次调用setState对状态队列进行合并时，之前对this.state的修改将会被忽略，造成无法预知的错误。

2、如果是由React引发的事件处理（比如通过onClick引发的事件处理），调用setState不会同步更新this.state,除此之外的setState调用会同步执行this.state。

原因：在React的setState函数实现中，会根据一个变量isBatchingUpdates判断是直接更新this.state还是放到队列中回头再说，而isBatchingUpdates默认是false，也就表示setState会同步更新this.state,但是，有一个函数batchedUpdates，这个函数会把isBatchingUpdates修改为true，而当react在调用事件处理函数之前就会调用这个batchedUpdates,造成的后果，就是由react控制的事件处理过程setState不会同步更新this.state。

react hooks中不存在此问题。

## 4、react组件通信是如何实现的？

1、父组件通过props向子组件进行通讯。

2、子组件向父组件通过props+回调的方式进行信息的传递

3、兄弟组件通信：通过结合上面两种方式进行通信

4、跨层级通信：Context设计目的是为了共享那些对于一个组件树而言是“全局”的数据

5、发布订阅模式：在哪都适用

6、全局状态管理工具：redux 和vue中vuex类似。

## 5、什么是高阶组件

高阶组件就是一个react组件包裹着另外一个react组件。

有点类似于vue中的proxy，java中的切面。可以对组件进行拦截，操作相应的数据后再传给原来的组件。

之后都会采用hooks

## 6、mixin、hoc、render props、react-hooks的优劣如何

React hooks优点：

1、简洁: React Hooks解决了HOC和Render Props的嵌套问题，更加简洁

2、解耦：React Hooks可以更方便地把UI和状态分离，做到更彻底的解耦。

3、函数友好：React Hooks为函数组件而生，解决了类组件的几大问题：this指向容易错误，分割在不同生命周期中的逻辑使得代码难以理解和维护，代码复用成本高。

缺点：react.memo不能完全替换shouldComponentUpdate。

## 7、Time Slice and Suspense

1、react在渲染的时候，不会阻塞现在的线程

2、如果你得设备足够快，你会感觉渲染是同步的

3、如果你的设备非常慢，你会感觉还算是灵敏的

4、虽然是异步渲染，但是你将会看到完整的渲染，而不是一个组件一行行的渲染出来

## 8、redux的工作流程

核心概念如下：

1、Store：保存数据的地方，你可以把它看成一个容器，整个应用只能有一个store

2、State：store对象包含的所有数据

3、Action：只是一个简单的json对象，type和payload作为键，用来触发state改变

4、Action creator：用来生成action的函数

5、Reducer: state被修改的地方

6、dispatch：通过dispatch发布action。

整个工作流程：

1、我们通常在请求完数据后，根据请求结果来dispatch一个action，请求结果成功，通常将数据添加到action中，失败的话，通常将msg添加到action中。

2、然后当分发一个action后，Reducer会根据action的类型，来处理state，以此改变state的状态。

3、一旦监听到state状态改变，就会更新view。

## 9、什么是virtual dom

Virtual DOM是对DOM的抽象，本质上是javascript对象。保存着一个个元素标签。

## 10、为什么需要Virtual DOM

DOM操作真正的问题DOM元素的位置、大小等改变，就会触发浏览器的重排或重绘，而这一操作是比较耗性能的。

在一个复杂的页面应用中，经常会涉及大量的DOM操作，如果每次都去操作真实的DOM，那么应用性能将非常低。而通过js操作虚拟DOM，最后一次性将所有的改变渲染成真实的dom，将大大地提升应用的性能。

## 11、虚拟dom的实现原理？

1、首先用javascript对象来模拟真实的DOM树，对真实的DOM进行抽象。

2、通过diff算法比较两个虚拟DOM树的差异，如果有差异就记录到一个对象里面。

其方式采用深度优先，同级比较，多余的节点进行批量操作，要么增加，要么删掉。这样可以将算法复杂度降低到O(n)。

3、patch算法，将两个虚拟DOM对象的差异应用到真正的DOM树上。

# TypeScript

## 1、什么是TypeScript

TypeScript是微软开发的一款开源的编程语言，它是JavaScript的超集，本质上是为JavaScript增加了静态类型声明。TypeScript最终也会被编译成JavaScript。

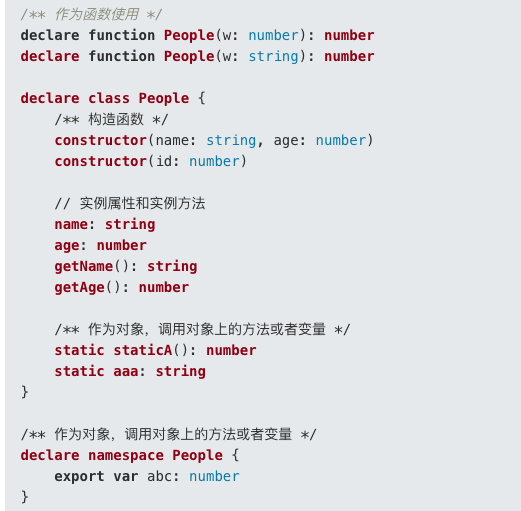
优点：

更好的可维护性和可读性。

在编译阶段就能发现大部分因变量类型导致的错误。

## 2、如何编写d.ts文件

通过declare关键字来声明函数或变量。



## 3、泛型

就是一个通用类型，传入的是什么类型，返回的就是什么类型。

比如我有个函数，有时候想处理number类型的数据，返回也是数组类型的数据，有时又想处理string类型的数据，那么就可以运用到泛型。