**CloudMusic项目复盘**

1. **零碎知识点：**
2. **拼接字符串：**

“${xx}” **便捷点在于不用再像以前那样用多对双引号拼接多对字符串了**

1. **模板字符串：**

ES6出现的，可以用来携带字符串，用反引号（`）表示。

用处1：保证多行字符串的原样式，即保留空格、换行等等。

用处2：配合${}使用，在字符串中嵌套变量或者函数运算

1. **ES6对象的简洁表示：**

例：const x='';

  const y=function(){};

  const z={

    x,

    y,

    "j":""

  }

这里就传入了三种类型：已有对象、已有方法、新定义对象

1. **路由传参：**

通过使用通配符向路由对应的组件传递参数，组件根据拿到的参数动态变换显示内容，同时地址栏的地址与传入的参数保持一致。

例：

  {

        path: "/recommend/",

        component: SuspenseComponent (RecommendComponent),

        routes: [

          {

            path: "/recommend/:id",

            component: SuspenseComponent (AlbumComponent)

          }

        ]

      }

Album组件通过props.match.params.id获取传入的歌单id，请求对应的歌曲资源渲染页面，同时路由也变成/recommend/:id，如下图

1. **导航组件的保留：**

想要在导航以后组件还在的话只要将导航组件声明在导向页面的上级路由中即可。

1. **扩大点击区域：**

position: relative;

    &:after {

      content: '';

      position: absolute;

      top: -10px; bottom: -10px; left: -10px; right: -10px;

    };

原理：通过没有宽度和高度的伪元素，设置他的top、bottom、left、right，这四个量的值都是相对父元素的。话又说来，父元素不就是需要扩大点击区域的元素吗，他的四维相对于父元素都放大了10px，那整体就胖了一圈。

1. **immutable的多层级查询：**

 {a:1}

immutableData.get('a') 得到1。

{a:{b:2}}

immutableData.getIn(['a', 'b']) 得到2。

1. **Swiper使用方法：**

  1.创建swiper容器，类名设置为‘slider-container’

  2.在useeffect中实例化swiper对象，并配置好初始化的参数，将类名传入到该对象中去，以供其定位

  3.在swiper容器中创建div，类名‘swiper-wrapper’，并在该div中插入轮播图图片数组，图片数组的使用需按照固定格式来，做好相关配置即可

1. **better-scroll使用步骤：**

1.创建一个div作为betterscroll的容器

2.创建一个bscroll对象，配置好bscroll的各项参数，并将该对象绑定到该div容器上

3.在useeffect中监听bscroll的各种状态，相应的做出相应的操作

1. **转发ref：**

在组件定义的同时包裹一层forwardRef，同时将外部传入的props和ref作为参数传入使用，形如：

const 组件名= forwardRef ((props, ref) => {组件内容}

1. **useImperativeHandle：**

useImperativeHandle限定了外部对ref定位到的元素的操作方式，只限定在useImperativeHandle提供了的操作方法范围内

本质是外部传入的ref只定位到函数组件，通过外部传入函数组件的ref调用useImperativeHandle给出的方法控制函数组件内部元素

需配合转发ref使用

1. **PropTypes：**

Proptypes帮助组件进行类型检查

例：

import PropTypes from 'prop-types';

class Greeting extends React.Component {

  render() {

    return (

      <h1>Hello, {this.props.name}</h1>

    );

  }

}

Greeting.defaultProps= {

  name: ”吴灏”

};

Greeting.propTypes = {

  name: PropTypes.string

};

[https://zh-hans.reactjs.org/docs/typechecking-with-proptypes.html#\_\_\_gatsby](https://zh-hans.reactjs.org/docs/typechecking-with-proptypes.html" \l "___gatsby)

1. **axios使用步骤：**

    1.创建一个axios实例并配置axios相关属性，export出去以供调用

    2.调用上一步创建好的axios实例，使用它的get，post...方法

3.可以设置拦截器在axios请求发出前和接收后对数据进行处理

export const baseUrl = 'http://111.229.243.7:3000/';

const axiosInstance = axios.create ({

  baseURL: baseUrl

});

axiosInstance.interceptors.response.use (

  res => res.data,

  err => {

    console.log (err, "网络错误");

  }

);

1. **immutable的set和fromJS方法：**

在immutable.js中，object经过fromjs函数以后默认转成map，array经过fromjs以后默认转成list

set方法在list和map中的使用：

   list：list.set(index, xx);

   map：map.set('key',xx)

1. **immutable的fromJS对于基本值类型的处理**

对于基本值类型无需进行fromjs转化即可存入immutable数据结构中了，因为fromJS的作用是将 JS 对象转换为 immutable 对象

1. **react-thunk源码分析**
2. **Array.from (xx)**

用以将类数组数据转化为真正的数组

1. **为什么是useRef而不是useState？**

1.相对于usestate，useref始终指向当前状态，而usestate则会在每次重新渲染的同时产生新的usestate，对于异步操作很有可能抓不到所需的实时值。

2.usestate在状态更新时会重新渲染组件，但是useref不会，这样有利于性能优化。

<http://www.yanwenbo.cn/topic/5e41427005c6ca503b4aa204>

https://zhuanlan.zhihu.com/p/102501434

https://segmentfault.com/a/1190000020435923?utm\_source=tag-newest

1. **函数防抖与节流**

**防抖：**

export const debounce = (func, delay) => {

  let timer;

  return function (...args) {

    if (timer) {

      clearTimeout (timer);

    }

    timer = setTimeout (() => {

      func.apply (this, args);

      console.log(this);//this指向调用闭包返回函数的对象

      clearTimeout (timer);

    }, delay);

  }

}

**通过闭包包含一个定时器，并且将要防抖的函数放到闭包中，调用定时器的延迟函数延迟执行。因为闭包，定时器常驻内存，且定时时间未到再次触发闭包会清零定时时间，重新定时启动，以达到防抖作用。**

**节流：**

function throttle(func, delay) {

let timer;

return function() {

let args = arguments;

if (!timer) {

timer= setTimeout(() => {

timer= null;

func.apply(this, args)

}, delay)

}

}}

**类似于防抖，与防抖的不同在于到了定时时间点就会执行，而不管触发了多少次**

<https://www.jianshu.com/p/c8b86b09daf0>

<https://blog.csdn.net/chenxi_li/article/details/101100467>

1. **useMemo和usecallback**

useCallback 能够帮我们在依赖不变的情况保持一样的函数引用，最大程度地节约浏览器渲染性能

**........**

1. **上下文：**

**执行栈：**执行栈用于存储代码执行期间创建的所有上下文，具有LIFO的特性

**全局执行上下文：**windows。var声明的对象全都挂到了windows下面

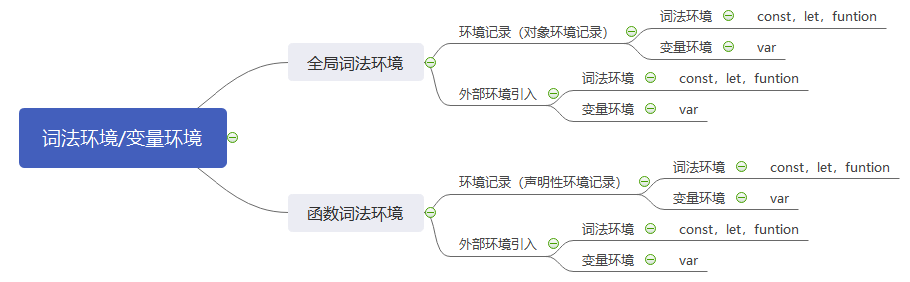
**函数执行上下文：**每当一个函数被调用时都会创建一个函数上下文；需要注意的是，同一个函数被多次调用，都会创建一个新的上下文

**执行上下文的创建和执行阶段：**

**创建阶段：**三件事--确定this、创建词法环境、创建变量环境

确定this。全局执行上下文中的this总是指向windows，而在函数执行上下文中，this的值取决于函数的调用方式，如果被一个对象调用，那么this指向这个对象。否则this一般指向全局对象window或者undefined（严格模式）

创建词法环境，创建变量环境。



**创建顺序：**this->全局词法环境环境记录->全局词法环境外部引入->全局变量环境环境记录->全局变量环境外部引入->函数词法环境环境记录->函数词法环境外部引入->函数变量环境环境记录->函数变量环境外部引入

**在执行上下文创建阶段，函数声明与var声明的变量在创建阶段已经被赋予了一个值，var声明被设置为了undefined，函数被设置为了自身函数，而let  const被设置为未初始化。**

**执行阶段：**根据创建的环境记录对其进行赋值

1. **apply,call,bind**

都是用来改变上下文的。

apply、call ：作用完全一样，只是接受参数的方式不太一样。call后面的参数与say方法中是一一对应的，而apply的第二个参数是一个数组，数组中的元素是和say方法中一一对应的

Bind：bind()方法会创建一个新函数，称为绑定函数，当调用这个绑定函数时，绑定函数会以创建它时传入 bind()方法的第一个参数作为 this，传入 bind() 方法的第二个以及以后的参数加上绑定函数运行时本身的参数按照顺序作为原函数的参数来调用原函数。

总结：

apply 、 call 、bind 三者都是用来改变函数的this对象的指向的；  
 apply 、 call 、bind 三者第一个参数都是this要指向的对象，也就是想指定的上下文；  
 apply 、 call 、bind 三者都可以利用后续参数传参；  
 bind 是返回对应函数，便于稍后调用；apply 、call 则是立即调用 。

<https://segmentfault.com/a/1190000018270750>

1. **JS中的&&**

只要“&&”前面是false，无论“&&”后面是true还是false，结果都将返“&&”前面的值;

只要“&&”前面是true，无论“&&”后面是true还是false，结果都将返“&&”后面的值;

1. **js中解构时的重命名**

const { rankList:list, loading,songsCount } = props;

1. **withRouter**

用withRouter包裹组件，向该组件传递props.history, props.location, props.match三个参数，以便于组件使用这三个参数

**注：被添加到路由节点的组件天生可以访问路由信息，即props.history, props.location, props.match；对于未被添加到路由节点的而组件需要手动包裹withrouter以便于组件对路由信息的访问。**

1. **Immutable.js的合并方法merge**

 x.merge(y):将y中的数据更新到x中，已有的覆盖，没有的新增，并最终返回一个新的对象

1. **splice和slice**

**slice(start,end)：**可从已有数组中返回选定的元素，返回一个新数组，包含从start到end（不包含该元素）的数组元素（该方法不会改变原数组）。**同时slice(start,end)可以作用于字符串的切割**

**start：**必须，规定从何处开始选取，如果为负数，规定从数组尾部算起的位置，-1是指最后一个元素。

**end：**可选（如果该参数没有指定，那么切分的数组包含从start倒数组结束的所有元素，如果这个参数为负数，那么规定是从数组尾部开始算起的元素）。

**splice（index,howmany,item1,...itemX）：**该方法向或者从数组中添加或者删除项目，返回被删除的项目。（该方法会改变原数组）

**index：**必须，整数，规定添加或者删除的位置，使用负数，从数组尾部规定位置。

**howmany：**必须，要删除的数量，如果为0，则不删除项目。

**item1,...itemX：**可选，向数组添加的新项目。

1. **isNaN()**

**isNaN(x)** 函数用于检查其参数x是否为非数字值。如果x为 NaN 或字符串、对象、undefined等非数字值则返回 true,否则返回 false

1. **e.stopPropagation()  /ˌprɒpəˈɡeɪʃn/**

**冒泡阻止事件：**让事件只在当前元素中生效而不向上传递到父祖元素上生效。常见的是控制某元素的onclick事件，保证只触发当前元素的点击事件而不触发父祖元素的相关事件。

1. **Touch类事件**

**touchstart:**手指触摸屏幕时触发

**touchendend:**手指离开屏幕时触发

**touchmove：**手指在屏幕中拖动目标元素时触发

**每个触摸事件被触发后，会生成一个event对象，event对象里额外包括以下三个触摸列表**

**touches:**当前屏幕上所有触点的列表

**targetTouches:**当前dom元素上触点的列表，尽量使用这个代替touches

**changedTouches:**涉及当前事件的触点的列表，尽量使用这个代替touches

**这些列表里的每次触摸由touch对象组成，touch对象里包含着触摸信息，主要属性如下：**

**clientX/clientY:**触摸点相对浏览器窗口的位置

**pageX/pageY:**触摸点相对于页面的位置

**screenX/screenY:**触摸点相对于屏幕的位置

**identifier:**touch对象的ID

**target:**当前的DOM元素

1. **getBoundingClientRect()方法**

**用以获取某元素的rect对象。**

**rect 是一个具有四个属性left、top、right、bottom的DOMRect对象**

**通过rect.left、rect.top等获取元素的边界值。**

1. **洗牌算法**

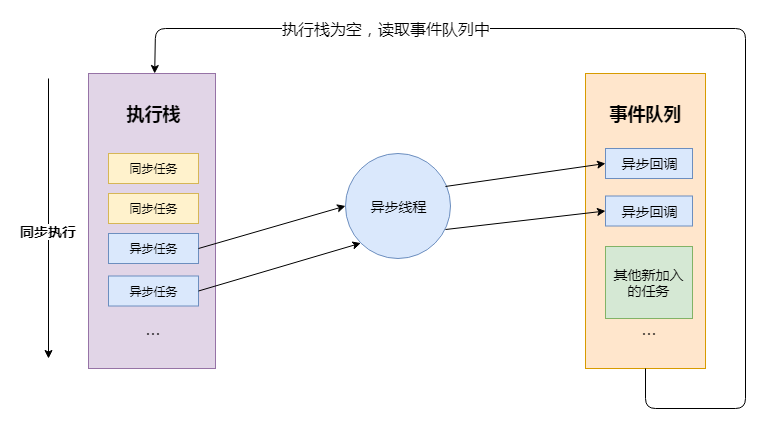
<https://blog.csdn.net/qq_26399665/article/details/79831490>

1. **Eventloop**

1.javascript 是单线程的

2.JS线程和UI线程是一对互斥线程， 当JS引擎线程执行时GUI渲染线程会被挂起，GUI更新则会被保存在一个队列中等待JS引擎线程空闲时立即被执行。

3.JS 分为同步任务和异步任务，同步任务都在JS引擎线程上执行，形成一个执行栈，事件触发线程管理一个事件队列，异步任务触发条件达成，将回调事件放到事件队列中，**执行栈中所有同步任务执行完毕**，此时JS引擎线程空闲，**系统会读取事件队列**，将可运行的异步任务回调事件添加到执行栈中，开始执行



**事件队列中的宏任务与微任务：**

**宏任务：**我们可以将每次执行栈执行的代码当做是一个宏任务（包括每次从事件队列中获取一个事件回调并放到执行栈中执行）， 每一个宏任务会从头到尾执行完毕，不会执行其他。GUI渲染线程会穿插各个宏任务的执行间隔之间执行.**主代码块，setTimeout，setInterval等，都属于**宏任务

**微任务：**穿插在宏任务与GUI渲染线程执行间隙间执行的任务。**Promise，process.nextTick等，属于**微任务

**宏任务-->微任务-->渲染-->宏任务-->微任务-->-->渲染．．．**

· 执行一个宏任务（栈中没有就从事件队列中获取）

· 执行过程中如果遇到微任务，就将它添加到微任务的任务队列中

· 宏任务执行完毕后，立即执行当前微任务队列中的所有微任务（依次执行）

· 当前宏任务执行完毕，开始检查渲染，然后GUI线程接管渲染

· 渲染完毕后，JS线程继续接管，开始下一个宏任务（从事件队列中获取）

**注：微任务的优先级是高于渲染的，promise的then是微任务，但是微任务是有任务队列的，异步执行完promise的then才会添加到微任务队列，所以如果异步没有执行完会直接执行下一个宏任务。**

<https://juejin.im/post/5d5b4c2df265da03dd3d73e5>

1. **React的双向绑定**

**M-V:**这个是react本就具备的能力，在Model层的state变化的同时立马重新渲染组件使得View层立马与Model层同步

**V-M:**需要在V层绑定onchange时间，触发事件的同时setstate从而更新M层，实现双向绑定

1. **JS中的作用域链**

在A作用域中使用的变量x，却没有在A作用域中声明（即在其他作用域中声明的），对于A作用域来说，x就是一个**自由变量**。

自由变量应该到**创建该函数的作用域中取**，而不是到父作用域中取，如果没找到继续往上层作用域找，直到找到为止。这一步步跨越路线又称为作用域链

1. **前端的mvc和react的mvvm**

**前端mvc:**

**M:ajax**

**V:html**

**C:js**

**react的mvvm:**

**M:redux,state,生命周期,方法**

**V:真实DOM+CSS**

**Vm:虚拟dom+diff**

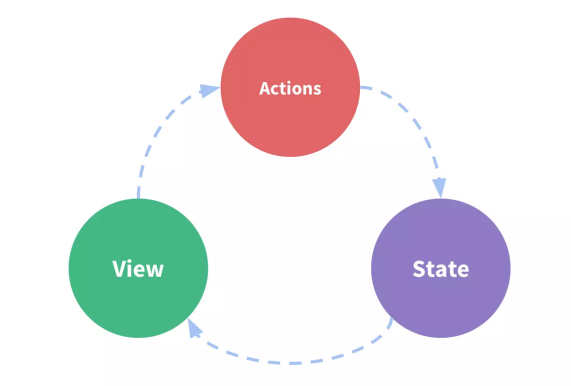
1. **Jsx的本质**

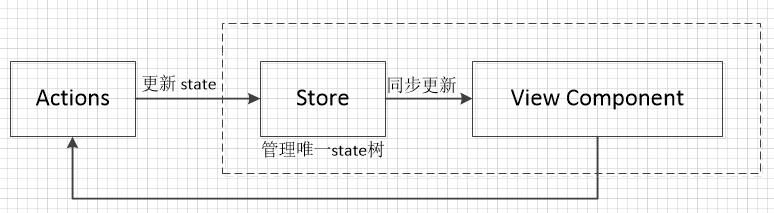
Jsx的本质是语法糖，其实质还是js。如jsx中的<div></div>其实就是React.createElement(‘div’)

1. **虚拟dom**

Jsx就是vdom，ReactDOM.render将vdom转化成真实的dom。

1. **Redux的单向数据流**

****



1. **setstate的异步**

**Setstate设置成异步的原因：**

连续的state变化如果是同步的会反复渲染造成性能浪费，设计成异步合并执行减少渲染次数

**具体案例：**

**handle() {**

**// 初始化 `count` 为 0**

**console.log(this.state.count) // -> 0**

**this.setState({ count: this.state.count + 1 })**

**this.setState({ count: this.state.count + 1 })**

**this.setState({ count: this.state.count + 1 })**

**console.log(this.state.count) // -> 0**

**}**

原因是异步事件会放入事件队列，待执行栈的任务都执行完成才会执行setstate。

**Setstate连续的相同调用会被合并成一次调用：**

上面案例最终count的值也只是1，三次被合并成一次执行。

1. **前端构建和打包的理由**

**1.传统方式css文件和js文件引用顺序的不对会引发问题，模块化就可以规避这个问题。**

**2.提高性能。多个js文件的合并减少http请求次数；文件压缩去空格降低文件体积。**

**3.编译。SCSS、ES6等都需要编译才能被浏览器识别。**

1. **Module、chunk、bundle分别是什么？有什么区别？**

**Module：**单个模块，如.js、.css、.png都是模块

**Chunk：**webpack在模块一来分析的时候，代码分割出来的代码块

**Bundle：**webpack最终打包出来的文件。

1. **Loader和plugin的区别**

**Loader：**用于对模块源码的转换，loader可以将文件从不同的语言（如TypeScript）转换为JavaScript，或者将内联图像转换为data URL

**Plugin：**目的在于解决loader无法实现的其他事，从打包优化和压缩，到重新定义环境变量

1. **Proxy**

**重要用途：**监听（参考：双向绑定，数据响应式）。

**格式：**

var proxy=new proxy(target,handle);

Target:代理对象

Handle:处理函数（get、set等13个函数）

1. **Reflect**

**作用：**

1.将一部分object的方法放到reflect中使用

2.修改一部分object方法的返回值使其更合理。如抛错变成false

3.让所有的object行为变成函数行为

4.让proxy调用reflect方法（reflect也是13中方法）

1. **Generator**

1.generator函数返回的不是函数体执行的结果，而是一个指针g，用来控制执行进度的

2.g可以调用next方法，每调用一次跳到最近的yield处

3.调用next会返回一个对象（value,done）

Value:yield后面的表达式的计算值

Done：generator是否执行完成的标志位

4.next的可以传参，参数取代的是上一个yield后面表达式的计算值

5.Generator函数函数体的内容的执行只能通过next函数进行

1. **Async和await**

Async是generator的语法糖，本质是在generator的外部报过了一个自动执行器spawn函数，不需要人为next去执行异步了，直接像同步函数一样执行完获得结果。

通过await卡住异步的点，执行执行栈中的其他任务去，异步执行完以后从await处继续往后执行，整个看起来写法就像是同步的写法，妙啊！

例：

  function timeout(ms) {

    return new Promise((resolve) => {

      setTimeout(resolve, ms);

    });

  }

**async** function asyncPrint(value, ms) {

    try{

**await** timeout(ms);

    }

    catch(err){

      console.log (err)

    }

    console.log(value)

  }

  asyncPrint('hello world', 50);

上面代码指定50毫秒以后，输出"hello world"。看起来就跟同步函数一样，牛逼啊！异步的最终选择！这里用try catch包裹起来，因为promise返回的不一定是resolve，也可能是reject

用promise搭配async去操作异步比较好，因为promise可以从外部插入函数到异步函数内，很爽就！

1. **New的原理剖析**

**New的作用：**

1.new会返回一个对象实例

2.访问构造函数原型上的属性

3.可以访问挂在构造函数this上的属性

**手动实现new：**

**前提：**向构造函数传入constructor，也就是constructor要在形参列表中。按照我的理解函数的constructor是指向函数的，所有constructor===function

1.定义一个空对象obj

2.将obj的\_proto\_指向构造函数的prototype

3.调用constructor.apply(obj,args)，将this绑定到对象实例上，同时向obj传递参数

4.判断constructor.apply(obj,args)的返回值res，如果res是对象就返回res，如果是值类型就返回obj。**需要说明的是obj是根据构造函数内部this绑定的值生成的对象，而res是传入的返回值，所以构造函数一般不要返回值，不然就失去意义了。**

**代码示例：**

function create(Con, ...args) {

  let obj = {}

  Object.setPrototypeOf(obj, Con.prototype)

  let result = Con.apply(obj, args)

  return result instanceof Object ? result : obj

}

function Test(name, age) {

  this.name = name

  this.age = age

}

Test.prototype.sayName = function () {

    console.log(this.name)

}

const a = create(Test, 'yck', 26)

console.log(a.name) // 'yck'

console.log(a.age) // 26

a.sayName() // 'yck'

1. **原型继承和class继承**

<http://www.ruanyifeng.com/blog/2010/05/object-oriented_javascript_inheritance.html>

**原型的继承分两部分进行：**

1. 继承挂在this上的变化的属性

下面的A.apply(this, arguments)其实相是执行了this.x=""，只不过this指向的对象是B中this所指向的对象。

    function A(){

      this.x="";

    }

    A.prototype.y="";

　　function B(){

　　　　A.apply(this, arguments);

　　}

var x=new B();

1. 继承挂在prototype上的不变的属性

    function A(){

      this.x="";

    }

    A.prototype.y="";

　　function B(){}

    B.prototype=new A();

    B.prototype.constructor=B;

    var x=new B();

**原型继承要点汇总：**

1. 对于不同的构造函数A,B，想让A继承B，那么让A.prototype=new B()，即A.prototype.\_proto\_===new B().\_proto\_===B.prototype,这样就可以继承B的protype上的所有属性了。this下面的直接在子函数函数体内调用’父函数.call(this,arguments)’就可以继承了
2. 对于构造函数，变化属性可以生命在构造函数的函数体内部，挂在this下面；对于不变的属性，应在函数体外部，挂到’构造函数.prototype’下面。
3. X.prototype指向发生改变的同时，应当同时改变X.prototype.constructor的指向对象为X,即X.prototype.constructor=X，否则会引起原型链紊乱。
4. 刚new出来的实例对象的内容与构造函数的prototype的内容一模一样。

**Class继承：**

    class A{

      constructor(){

        this.x="";

      }

      y(){}

    }

    class B extends A{

      constructor(){

        super();

      }

    }

    let xx=new B();

    b.y();

1. **Arguments的一个坑**

Arguments[0]()等同于argument.xx(),xx是[0]中存储的数据，这里this会指向arguments。也就是说一旦有x[number]就要能想到转化为x.number对象。

1. **如何让A标签不跳转页面而是下载**

声明content-type为stream即可

1. **In运算符**

In运算符可以判断对象是否包含某种属性。

**例：**”属性名” in 对象 返回值是一个boolean值，true就是包含，false就是不包含。

1. **浏览器的事件机制**

1.浏览器先执行捕获事件，后执行冒泡事件

2.AddEventListener(事件名，处理函数，模式)

3.模式是布尔值，改变模式可以改变执行顺序。true为先执行捕获，false为先执行冒泡

4.stopPropagation可以用来阻止事件的传播（捕获/冒泡）。

1. **事件代理**

对于子节点较多或者子节点动态变化的情况，监听每一个子节点是不现实的。可以通过监听父节点来实现对子节点的事件代理。

1. **原始类型：string，number，boolean，null，undifined，symbol**
2. **Type of和instanceof**

**判断原始类型用type of比较好**

typeof 1 // 'number'

typeof '1' // 'string'

typeof undefined // 'undefined'

typeof true // 'boolean'

typeof Symbol() // 'symbol'

**判断对象类型用instanceof比较好**

const Person = function() {}

const p1 = new Person()

p1 instanceof Person // true

var str = 'hello world'

str instanceof String // false

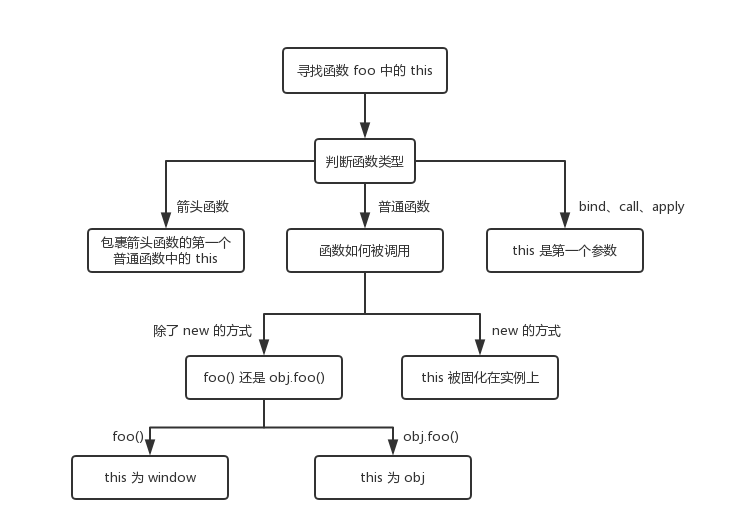
var str1 = new String('hello world')

str1 instanceof String // true

1. **类型转换**



1. **this的指向**



1. **闭包**

闭包不一定非要返回一个函数，只要是在A函数中定义了函数B，并且B中使用到了A中的变量，B就是个闭包。

为什么会有闭包？说到底是作用域链，由于闭包函数对自由变量的需求导致应该被销毁的上下文保留了下来，从而常驻内存，产生了闭包。**闭包所需求的上下文环境不会自动销毁，需要手动赋值闭包依赖变量所在函数为null才能清除对应的上下文。**

1. **==比较符**

对于 == 来说，如果对比双方的类型****不一样****的话，就会进行****类型转换****

假如我们需要对比x和y是否相同，就会进行如下判断流程：

1. 首先会判断两者类型是否相同。相同的话就是比大小了
2. 类型不相同的话，那么就会进行类型转换
3. 会先判断是否在对比null和undefined，是的话就会返回true
4. 判断两者类型是否为string和number，是的话就会将字符串转换为number
5. 判断其中一方是否为boolean是的话就会把boolean转为number再进行判断
6. 判断其中一方是否为object且另一方为string、number或者symbol，是的话就会把object转为原始类型再进行判断
7. **深浅拷贝**

**浅拷贝：**

1. **object.assign()**

let a = {

  age: 1

}

let b = Object.assign({}, a)

a.age = 2

console.log(b.age) // 1

1. **ES6三点运算符**

let a = {

  age: 1

}

let b = { ...a }

a.age = 2

console.log(b.age) // 1

浅拷贝只解决了第一层的问题，如果接下去的值中还有对象的话，就需要深拷贝登场了

**深拷贝：**

**1.JSON.parse(JSON.stringify(object)) 有局限性**

**2.lodash的深拷贝函数**

1. **原型**

1.一切（引用类型）都是对象，对象是属性的集合

2.对象都是通过函数创建的

3.每个函数都有一个属性叫做prototype，这个prototype的属性值是一个对象（属性的集合，再次强调！），默认的只有一个叫做constructor的属性，指向这个函数本身。

4.每个函数function都有一个prototype，同时每个对象都有一个\_\_proto\_\_，且每个对象的\_\_proto\_\_属性，指向创建该对象的函数的prototype

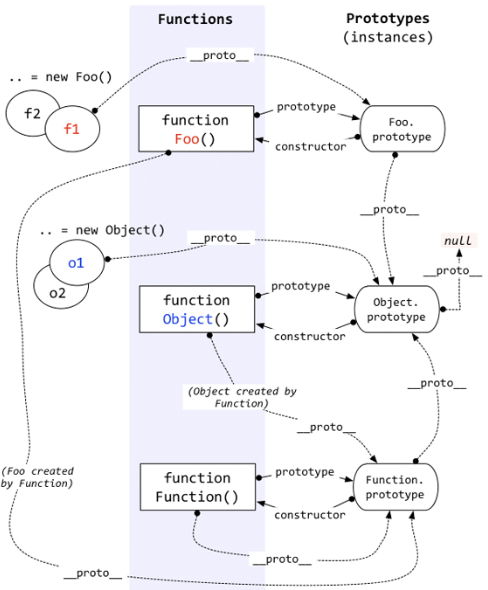
5.Object.prototype确实一个特例——它的\_\_proto\_\_指向的是null

6.函数也是一种对象：

例：var fn=new Function(“x”,”y”,”return x+y;”);

Console.log(fn(5,6));

7.Function的\_proto\_指向自身的prototype



**总结：**

1. **\_proto\_一定是指向prototype的（Object.prototype的\_proto\_除外，指向null）**
2. **对象是没有prototype属性的，函数才有**
3. **对象的\_proto\_指向创建该对象的函数的prototype**
4. **所有函数（除Object之外）的prototype的\_proto\_指向Object的prototype**
5. **所有函数（包含Function）自身的\_proto\_指向Function的prototype**
6. **Instanceof的原理：**

**A instanceof B**

**A.\_proto\_.\_proto\_.\_proto\_...一直往下挖，直到发现其等于B.Prototype**

**返回true，否则返回false**

访问一个对象的属性（键值对和方法）时，先在基本属性中查找，如果没有，再沿着\_\_proto\_\_这条链向上找，这就是**原型链**。

function Foo(){}

var f1 = new Foo();

f1.a=10;

Foo.prototype.a=100;

Foo.prototype.b=200;

console.log(f1.a);//10

console.log(f1.b);//200

<https://www.cnblogs.com/wangfupeng1988/p/3977987.html>

1. **var、let、const的区别**

1.函数提升优先于变量提升，函数提升会把整个函数挪到作用域顶部，变量提升只会把声明挪到作用域顶部

2.var存在提升，我们能在声明之前使用。let、const 因为暂时性死区的原因，不能在声明前使用

3.var在全局作用域下声明变量会导致变量挂载在 window 上，其他两者不会

4.let和const作用基本一致，但是后者声明的变量不能再次赋值

1. **Promise**

固定格式：

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

  // ... some code

  if (/\* 异步操作成功 \*/){

    resolve(value);

  } else {

    reject(error);

  }});

promise.then(function(value) {

   success},

//function(error) {

  // failure}

);

Then中的第二个参数是可以省略的，一般来说，不要在then方法里面定义 Reject 状态的回调函数（即then的第二个参数），总是使用catch方法

promise

.then(function(data) {})

.catch(function(err) {});

**Promise使用步骤：**

1. 声明一个promise对象，其中包含了一个函数，该函数具有
2. 在promise体中声明包含两个形参resolve和reject的函数，分别用以处理异步请求的成功和失败。
3. 调用promise的then和catch，分别向他们传入回调函数，处理异步操作后返回的成功和失败的结果

**Promise的链式调用：**

getJSON("/posts.json").then(function(json) {

return json.post;}).then(function(post) {

// ...});

注★：后面每一个then获得的resolve和reject参数都是在、上一个then中的resolve和reject函数的返回值

Promise.all()方法用于将多个 Promise 实例，包装成一个新的 Promise 实例。

const p = Promise.all([p1, p2, p3]).then(x(),y());

只有当p1,p2,p3都成功才会调用x（），否则调用y（），且是将p1,p2,p3执行完后将他们各自resolve或者reject的参数组合成数组传递给x（）或者y（）。

比较不错的例子： <https://blog.csdn.net/u012045958/article/details/80635151>

1. **跨域**

**定义：**发送和接收双方协议、域名、端口只要有一个不一样就会形成跨域。

**目的：**预防CSRF攻击。（CSRF:利用用户登录状态二发起的恶意请求）

**解决手段：**

**①JSONP：**利用script标签没有跨域限制的特点，将src指向目标地址，且通过在url中插入’callback=xxx’参数提供回调函数来接收数据。

**②CORS:**后端配置Access-Control-Allow-Origin

1. **检测对象类型的方法：**

Object.prototype.tostring.call()

**例：**Object.prototype.toString.call([1,2,3])返回[object Array]

1. **Ajax原理剖析**

**原理：XMLHttpRequest**对象向服务器发送异步请求

**XMLHttpRequest的属性和方法：**





**步骤：**

1. 创建XMLHttpRequest对象  
    var xmlHttp;

if (window.XMLHttpRequest) { //非IE

xmlHttp = new XMLHttpRequest();

} else if (window.ActiveXObject) { //IE

xmlHttp = new ActiveXObject("Microsoft.XMLHTTP")

}

1. 设置onreadystatechange回调函数

xmlHttp .onreadystatechange=function(){

if (xmlHttp .readyState==4 && xmlHttp .status==200){

document.getElementById("myDiv").innerHTML=xmlHttp .responseText;

}

}

1. 调用open方法设置请求方式、url以及是否异步

xmlhttp.open("GET",url,true);

1. 调用send方法发送异步请求

xmlhttp.send();

1. **垃圾回收**
2. **手写call**

步骤：

1. 在Funciton.prototype上声明一个mycall方法
2. 定义mycall的执行内容：

①为传入mycall的对象obj创建一个空方法x

②将this赋值给x，即让x方法变成调用call的函数

③执行obj.x()

④删除方法x

1. **React的组件懒加载**
2. **柯里化**
3. **Math.round、math.ceil、math.floor的区别**

Math.round:四舍五入

Math.ceil:进一

Math.floor:去尾

1. **Websocket**
2. **Usecontext**

<https://juejin.im/post/5ceb37c851882520724c7504>

**Context：**

**context的使用步骤：**

1.export const x=React.createContext(初始值)创建一个context

2.在顶层父组件声明

<x.Provider value={{name:”wuhao”}}>

...子组件

</x.Provider>

用provider包裹子组件，从而子组件可以拿到context中存放的值

3.如果是class组件，可以选择先引入父组件中声明的context对象x，然后在子组件内部声明static contextType=x，随即可以通过this.context.name调用父组件中声明的context中保存的值了

4.如果是function组件，则必须用

<x.Consumer>

(name)=>{

...子组件内容

}

</x.Consumer>

包裹子组件的内容,并在子组件中调用context中预存的值了

**Usecontext：**

**usecontext是对传统context使用的改良，具体步骤如下：**

1. export const x=React.createContext(初始值)创建一个context
2. 在顶层父组件声明

<x.Provider value={{name:”wuhao”}}>

...子组件

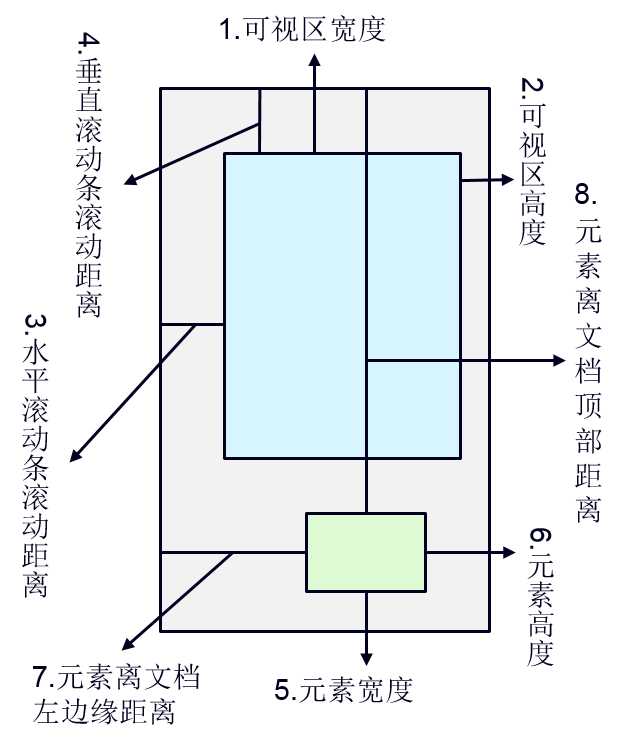
</x.Provider>

用provider包裹子组件，从而子组件可以拿到context中存放的值

1. 在子组件中import父组件中声明的context X,并在函数组件内部声明usecontext（x）,获取context中存放的值，usecontext会返回读取的context的值，可以通过解构直接获取，如：let {name}=useContext(x),这样子组件中就可以使用了
2. **useReducer**
3. **React进行异步请求在哪个生命周期进行**
4. **手写promise**
5. **如何获取设备类型**

通过navigator.userAgent判断。Useragent中包含了日常判断所需要的各种信息。

1. **浏览器本地存储**
2. **Import和require的区别**
3. **点击穿透和滑动穿透**
4. **排序算法**
5. **图片懒加载原理**



**图片加载触发条件：**

元素距离文档顶部的距离-垂直滚动距离-可视区高度<0 &&

元素距离文档顶部的距离+元素宽度-垂直滚动距离>0 &&

元素离文档左边远距离+元素宽度-水平滚动距离>0 &&

元素离文档左边远距离-水平滚动距离-可视区宽度<0

元素距离文档顶部的距离:

元素离文档左边远距离:

水平滚动距离window.pageXOffset

垂直滚动距离:window.pageYOffset

可视区高度:window.innerHeight

可视区宽度:window.innerWidth

1. **清除浮动**

**常用方法：**

1. 在浮动元素下方添加一个空div，并设置样式clear：both
2. 自定义Clearfix类：添加伪元素，设置display为block，并添加clear：both

.clearfix::after {

content: "";

display: block;

clear: both;

}

这里其实跟第一种差不多，通过伪元素模拟出了一个空的div，并设置了claer：both

1. 父元素添加overflow:hidden或者overflow:auto
2. **移动端设备兼容性**
3. 设置meta的name为viewport，并设置content为相关属性
4. 通过media进行自适应,@media 设备类型 and （宽度阈值){样式代码}

宽度阈值：

max-width:xx 代表在屏幕宽度小于xx的时候使用该部分样式定义

Min-width:xx 代表在屏幕宽度大于xx的时候使用该部分样式定义

对媒体的限制条件可多余一项，用and连接

1. **常用布局方式**

**静态布局、流式布局(百分比布局)、弹性布局、响应式布局：**

1. 静态布局

特点：最外层宽高固定值，超出部分滚动条

手段：设置max-width或者min-width

1. 流式布局（百分比布局）

特点：随着屏幕的改变页面布局没有发生大变化，可以进行适配调整。

手段：左侧固定，右侧自适应 或者 左右固定，中间自适应 元素宽高全部百分比

1. 弹性布局（flex布局）

不多说

1. 响应式布局

特点：自动适应不同的设备

手段：通过media不同分辨率设置不同的流式布局方案