**CloudMusic项目复盘**

1. **遇到的问题：**
2. **零碎知识点：**
3. **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. **Styled-component声明全局样式：**

export const GlobalStyle = createGlobalStyle``

通过createGlobalStyle声明全局样式，然后在项目根文件中导入即可

function App() {

  return (

    <Provider store={store}>

      <HashRouter>

        <GlobalStyle></GlobalStyle>

        <IconStyle></IconStyle>

        { renderRoutes(routes) }

      </HashRouter>

    </Provider>

  )

}

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. **CSS样式最高优先级！Important**

例：

font-family: "iconfont" !important;

！important可以突破css引用的自然优先级，优先使用！Important标记的样式

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. **’>’选择器**

'>'代表后代第一层选择器

1. **Flex布局常用属性：**

**主轴方向：flex-direction**

**flex-direction**: row | row-reverse | column | column-reverse;

row（默认值）：主轴为水平方向，起点在左端。

row-reverse：主轴为水平方向，起点在右端。

column：主轴为垂直方向，起点在上沿。

column-reverse：主轴为垂直方向，起点在下沿。

**项目溢出换行：flex-wrap**

**flex-wrap**: nowrap | wrap | wrap-reverse;

nowrap（默认）：不换行。

wrap：换行，第一行在上方。

wrap-reverse：换行，第一行在下方。

**★flex-flow：flex-flow**属性是flex-direction属性和flex-wrap属性的简写形式，默认值为row nowrap。

**★主轴对齐方式：justify-content**

**justify-content**: flex-start | flex-end | center | space-between | space-around;

flex-start（默认值）：左对齐

flex-end：右对齐

center： 居中

space-between：两端对齐，项目之间的间隔都相等。

space-around：每个项目两侧的间隔相等。所以，项目之间的间隔比项目与边框的间隔大一倍。

**★交叉轴对齐方式：align-items**

**align-items**: flex-start | flex-end | center | space-between | space-around | stretch;

flex-start：交叉轴的起点对齐。

flex-end：交叉轴的终点对齐。

center：交叉轴的中点对齐。

baseline: 项目的第一行文字的基线对齐。

stretch（默认值）：如果项目未设置高度或设为auto，将占满整个容器的高度。

**项目属性：**

**flex-grow**属性定义项目的放大比例，默认为0，即如果存在剩余空间，也不放大。flex-grow: <number>

**flex-shrink**属性定义了项目的缩小比例，默认为1，即如果空间不足，该项目将缩小。flex-shrink: <number>

**flex-basis**属性定义了在分配多余空间之前，项目占据的主轴空间（main size）。浏览器根据这个属性，计算主轴是否有多余空间。它的默认值为auto，即项目的本来大小。flex-basis: <length> | auto;

**★flex**属性是flex-grow, flex-shrink 和 flex-basis的简写，默认值为0 1 auto。后两个属性可选。flex: none | [ <'flex-grow'> <'flex-shrink'>? || <'flex-basis'> ]

**★align-self**属性允许单个项目有与其他项目不一样的对齐方式，可覆盖align-items属性。默认值为auto，表示继承父元素的align-items属性，如果没有父元素，则等同于stretch。align-self: auto | flex-start | flex-end | center | baseline | stretch;

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

**13.元素高度百分比设置无效的问题**

元素的高度默认是缺省值，所以特别要注意当高度设置为百分比时，父元素的高度要确定，否则设置无效，尤其是在height：100%时，最终可能会追溯到设置html，body的高度为100%

**14.display属性对布局带来的影响**

**块级元素：**每个块级元素通常都会独占一行或者是多行，可以对其单独设置高度,宽度以及对齐等属性。

**特点：**

块级元素会独占一行

高度，行高，外边距和内边距都可以单独设置

宽度默认是容器的100%

可以容纳内联元素和其他的块级元素

**行内元素：**不占有独立的区域，仅仅依靠自己的字体大小或者是图像大小来支撑结构。一般不可以设置宽度，高度以及对齐等属性。

**特点：**

和相邻的行内元素在一行上

高度和宽度无效，但是水平方向上的padding和margin可以设置，垂直方向上的无效

默认的宽度就是它本身的宽度

行内元素只能容纳纯文本或者是其他的行内元素（a标签除外）

#### **行内块级元素：**

和相邻的行内元素（行内块）在一行上，但是中间会有空白的间隙

默认的宽度就是本身内容的宽度

高度，行高，内边距和外边距都可以设置

**15.react-lazyload使用方法**

1.在图片的外面包裹一层LazyLoad，并设置placeholder属性为要懒加载时显示的内容（可以是图片或者其他）

2.向scroll的onscroll属性传入执行函数，当滑动的时候执行该函数，即传入forcecheck函数，作用是解除懒加载状态显示图片，实现手段是。。。有待考证

forcecheck

**16.占位容器的技巧**

容器的margin\padding使用百分比时是以父元素的width为基准的，所以'padding-top: 100%'就是父元素的width，同时不能使用margin的理由是如果用了margin容器就没有高度了，但是用padding有，因为默认的元素的高度是content的高度+padding的高度，如果用了margin，高度会受到挤压，反之padding则不会

**17.利用背景阴影做衬底的技巧**

background: linear-gradient(hsla(0,0%,43%,.4),hsla(0,0%,100%,0));

hsla(hue, saturation, lightness, alpha)

hue- 色相:范围0-360，其中0 (或 360) 为红色, 120 为绿色, 240 为蓝色

saturation - 饱和度：定义饱和度; 0% 为灰色， 100% 全色

lightness - 亮度：定义亮度 0% 为暗, 50% 为普通, 100% 为白

alpha - 透明度：定义透明度 0（透完全明） ~ 1（完全不透明）

linear-gradient(direction, color-stop1, color-stop2, ...)

direction：  用角度值指定渐变的方向（或角度）。

color-stop1, color-stop2,...：用于指定渐变的起止颜色。

**18.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

**19.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. **CSS动画的编写**

keyframes是专门用来写动画用的. "from" 和 "to"，等同于 0% 和 100%，0% 是动画的开始，100% 是动画的完成.

animation要规定至少两个属性：动画名称、动画时长

步骤：

1.编写keyframes

@keyframes myfirst

{

from {background:red;}

to {background:yellow;}

}

2.在animation属性调用keyframes

div

{

animation:myfirst 5s;

}

1. **CSS过渡属性transition使用：**

语法：

transition: property duration timing-function delay;

[property](https://links.jianshu.com/go?to=http://www.runoob.com/cssref/css3-pr-transition-property.html" \t "https://www.jianshu.com/p/_blank):规定应用过渡的 CSS 属性的名称(默认为all)。(可选)

[duration](https://links.jianshu.com/go?to=http://www.runoob.com/cssref/css3-pr-transition-duration.html" \t "https://www.jianshu.com/p/_blank):定义过渡效果花费的时间。默认是 0s。（必需）

[timing-function](https://links.jianshu.com/go?to=http://www.runoob.com/cssref/css3-pr-transition-timing-function.html" \t "https://www.jianshu.com/p/_blank):规定过渡效果的时间曲线。默认是 "ease"函数。(可选)

[delay](https://links.jianshu.com/go?to=http://www.runoob.com/cssref/css3-pr-transition-delay.html" \t "https://www.jianshu.com/p/_blank):规定过渡效果何时开始。默认是 0s.(可选)

[timing-function](https://links.jianshu.com/go?to=http://www.runoob.com/cssref/css3-pr-transition-timing-function.html" \t "https://www.jianshu.com/p/_blank)表：

| 值 | 描述 |
| --- | --- |
| linear | 规定以相同速度开始至结束的过渡效果，即匀速。（等于 cubic-bezier(0,0,1,1)）。 |
| ease | 规定慢速开始，然后变快，然后慢速结束的过渡效果（cubic-bezier(0.25,0.1,0.25,1)）。 |
| ease-in | 规定以慢速开始的过渡效果（等于 cubic-bezier(0.42,0,1,1)）。 |
| ease-out | 规定以慢速结束的过渡效果（等于 cubic-bezier(0,0,0.58,1)）。 |
| ease-in-out | 规定以慢速开始和结束的过渡效果（等于 cubic-bezier(0.42,0,0.58,1)）。 |
| cubic-bezier(n,n,n,n) | 在 cubic-bezier 函数中定义自己的值。可能的值是 0 至 1 之间的数值。 |

例子：

div

{

transition: width 2s;

}

1. **CSS transform属性**
2. **CSS xx:nth-child(y)属性**

规定xx元素的父元素的第y个子元素的属性

例：这里就是让包裹2的span标签字体变红

<style>

div:nth-child(2){

Color:red

}

</style>

<div>

<span>1</span>

<span>2</span>

</div>

1. **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。

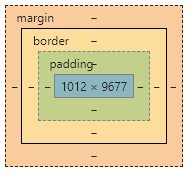
2.usestate在状态更新时会重新渲染组件，但是useref不会，这样有利于性能优化。

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

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

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

1. **offset：元素盒模型的宽度**
2. **box-sizing：**



border-box：宽高的范围是border+padding+content

content-box:宽高的范围是content

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**

seCallback 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

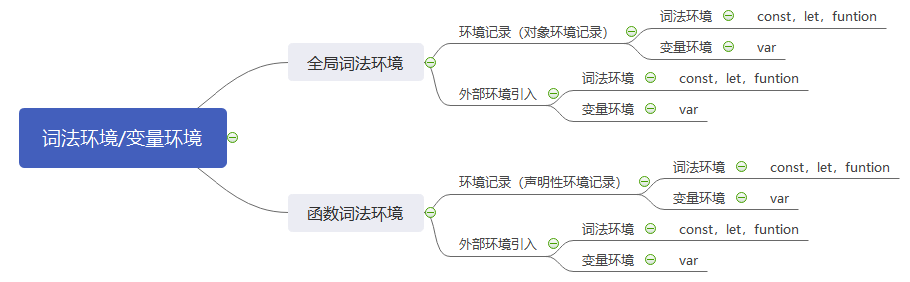
**函数执行上下文：**每当一个函数被调用时都会创建一个函数上下文；需要注意的是，同一个函数被多次调用，都会创建一个新的上下文

**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. **单位vh、和vw**

vh vw是不包含页面滚动条的视窗宽度(innerwidth)，%包含了滚动条的宽度在里面了(outerwidth)

vw：1vw等于视口宽度的1%。

vh：1vh等于视口高度的1%。

1. **伪类和伪元素**

伪类： :hove、:activer、:nth-child(n)

伪元素： ::before、::after

其中:nth-child(n)规定了使用该样式的元素的父元素的第n个子元素使用该样式

当n=odd时，标志着奇数位使用该样式；当n=even时，标志着奇数位使用该样式

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

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。

**tem1,...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()**

**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. **Translate偏移时的相对对象**

**无论是在动画中还是在react-transition-group中translate都是相对于初始位置做移动。**

**也就是说如果你在20%的时候执行translate（50px,0）向右平移了50px**

**在30%的时候执行translate（0,0）则会向左平移50px**

**因为你的移动的相对对象永远是初始位置**

1. **sass中父选择器的妙用**

**<div className="background">**

**<div className="background layer"></div>**

**.background {**

**position: absolute;**

**left: 0;**

**top: 0;**

**width: 100%;**

**height: 100%;**

**z-index: -1;**

**opacity: 0.6;**

**filter: blur(20px);**

**&.layer {**

**background: ${style["font-color-desc"]};**

**opacity: 0.3;**

**filter: none;**

**}**

**}**

backgroud是公用样式，.&layer是同时声明background和layer的元素才享有的独有样式。

1. **dangerouslySetInnerHTML**

**react的dangerouslySetInnerHTML在一个dom里直接筛入一段纯html**

**格式：dangerouslySetInnerHTML={{ \_\_html: content }} content为元素显示的内容**

**该操作是不安全的，会引发XSS攻击，所以他才叫做‘dangerouslySetInnerHTML’ 是有风险的  能不用则不用**

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. **clientWidth、offsetWidth、scrollWidth的区别**

**clientWidth = width+左右padding**

**offsetWidth = width + 左右padding + 左右boder**

**scrollWidth：获取指定标签内容层的真实宽度（可视区域宽度+被隐藏区域宽度）。**

1. **洗牌算法**

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

1. **SVG**

**首先需要定义一个<svg></svg>标签来作为svg图像的容器，后面可以在这个容器中画线画圆画正方形...**

<svg width={radius} height={radius} viewBox="0 0 100 100" version="1.1" xmlns="http://www.w3.org/2000/svg">

    <circle r="50" cx="50" cy="50" fill="transparent"/>

    <circle r="50" cx="50" cy="50" fill="transparent"

   strokeDasharray={dashArray}strokeDashoffset={dashOffset}/>

</svg>

**viewBox：一个用来适配svg内容和这个容器本身的工具，viewbox的大小是相对于svg的内容的，并不是相对于svg容器。先用viewbox将将好装下了svg的内容 ，再由svg自动适配viewbox，使得viewbox能正好完整展示在svg中。具体过程就是一个可变大小的框子（viewbox），去框svg的内容，然后svg容器又是个框子，不过是固定大小的，这个时候viewbox调整自己的大小（svg的内容与viewbox同步变大变小），正好能套进svg容器中，O了！ viewBox="x, y, width, height" x:左上角横坐标，y:左上角纵坐标，width:宽度，height:高度**

<circle r="50" cx="50" cy="50" fill="transparent"

        strokeDasharray={dashArray}

        strokeDashoffset={dashOffset}/>

**Circle：圆**

**fill：填充色**

**r:半径**

**cx：x坐标**

**cy：y坐标**

**transparent：透明色**

**strokeDasharray：用于描绘出虚线边框  格式：strokeDasharray={线长度，缺口长度， 线长度，缺口长度...}**

**strokeDasharray内容的数值个数必须是偶数个 否则自动复制原内容个数自动翻倍**

**例：321 -> 321321  这样321321就是一个循环周期了**

**strokeDashoffset：起始点的偏移量  偏移方向：正左负右**

<https://segmentfault.com/a/1190000009226427>

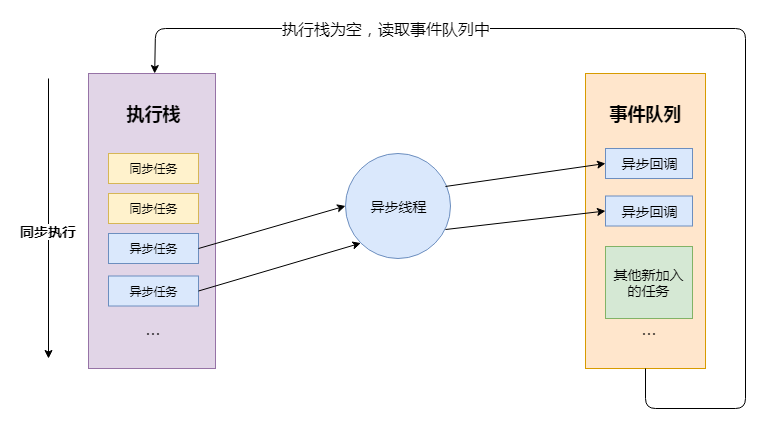
<https://www.cnblogs.com/daisygogogo/p/11044353.html>

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:**