2022前端面试题

html

从浏览器地址栏输入 url 到请求返回发生了什么?

浏览器 URL 解析 -> DNS解析(获取ip地址, 先缓存, 没有根域名服务器查 -> com顶级域名 -> 权威域名) -> TCP链接(http三次握手, https多了一个TLS加密协议的握手过程) -> http请求 -> 服务器响应 -> 浏览器开始渲染 -> 四次挥手关闭链接

问: TCP 连接为啥三次握手?

客户端和服务端要进行可靠传输，那么就需要**确认双方的接收和发送能力。**第一次握手可以确认客服端的**发送能力**，第二次握手，确认了服务端的**发送能力和接收能力**，所以第三次握手才可以确认客户端的**接收能力。**不然容易出现丢包的现象

CSS

JS

如何判断变量是否为数组?

```js
Array.isArray(arr);
arr instanceof Array;
Object.prototype.toString.call(arr) === '[object Array]';
```

根据 0.1+0.2！== 0.3，讲讲 IEEE 754 ，如何让其相等?

原因:

- **进制转换:** js数字计算时, 会转换为二进制. 最大存储 **53位** 有效数字, 大于会被截掉. 导致精度丢失
- **对阶运算:** 由于指数位数不相同，运算时需要对阶运算，阶小的尾数要根据阶差来右移（0舍1入），尾数位移时可能会发生数丢失的情况，影响精度

如何让其相等,

- 字符串相加

```js 复制代码
// 字符串数字相加
var addStrings = function (num1, num2) {
  let i = num1.length - 1;
  let j = num2.length - 1;
  const res = [];
  let carry = 0;
  while (i >= 0 || j >= 0) {
    const n1 = i >= 0 ? Number(num1[i]) : 0;
    const n2 = j >= 0 ? Number(num2[j]) : 0;
    const sum = n1 + n2 + carry;
    res.unshift(sum % 10);
    carry = Math.floor(sum / 10);
    i--;
    j--;
  }
  if (carry) {
    res.unshift(carry);
  }
  return res.join("");
};

function isEqual(a, b, sum) {
  const [intStr1, deciStr1] = a.toString().split(".");
  const [intStr2, deciStr2] = b.toString().split(".");
  const inteSum = addStrings(intStr1, intStr2); // 获取整数相加部分
  const deciSum = addStrings(deciStr1, deciStr2); // 获取小数相加部分
  return inteSum + "." + deciSum === String(sum);
}

console.log(isEqual(0.1, 0.2, 0.3)); // true
```

new 实现

new 实现了哪些功能?

- 能访问构造函数的属性
- 把这个对象的构造原型(__proto__)指向函数的原型对象 prototype ， 并绑定this
- 能访问函数原型中的属性.
- 如果构造函数带返回值. 返回类型如果是对象. 则返回这个对象. 否则还是实例对象.


```
function New(fn) [
  var obj = {}
  obj.__proto__ = fn.prototype;
  var res = fn.apply(obj, [...arguments.slice(1)])
  return typeof res === 'object' ? res : obj;
]
```

Promise.all 的实现


```
// Promise.all([A, B])
Promise.all = function(arr) {
    let result = []
    return new Promise((resolve, reject)=>{
        let remaining = arr.length
        function res(val, i) {
            try {
                if (val && (typeof val === 'function' || typeof val === 'object')) {
                    const {
                        then
                    } = val;
                    // 遗漏
                    if (typeof then === 'function') {

                        then.call(
                            val,
                                function(value) {
                                res(value, i)
                            },
                                reject
                        );
                        return;
                    }
                }
                result[i] = val;
                if (--remaining === 0) {
                    resolve(result)
                }
            } catch(error) {
                reject(error)
            }
```

```
        }
        for (let i = 0; i < arr.length; i++) {
            res(arr[i], i)
        }
    })
}
```

http

GET 和 POST 的区别。

- 从**缓存**的角度，GET 请求会被浏览器主动缓存下来，留下历史记录，而 POST 默认不会。
- 从**编码**的角度，GET 只能进行 URL 编码，只能接收 ASCII 字符，而 POST 没有限制。
- 从**参数**的角度，GET 一般放在 URL 中，因此不安全，POST 放在请求体中，更适合传输敏感信息。
- 从**幂等性**的角度，GET 是幂等的，而 POST 不是。(幂等表示执行相同的操作，结果也是相同的)
- 从 **TCP** 的角度，GET 请求会把请求报文一次性发出去，而 POST 会分为两个 TCP 数据包，首先发 header 部分，如果服务器响应 100(continue)， 然后发 body 部分。(火狐浏览器除外，它的 POST 请求只发一个 TCP 包)

简要概括一下 HTTP 的特点？HTTP 有哪些缺点?

特点:

1. **灵活可扩展**，主要体现在两个方面。一个是语义上的自由，只规定了基本格式，比如空格分隔单词，换行分隔字段，其他的各个部分都没有严格的语法限制。另一个是传输形式的多样性，不仅仅可以传输文本，还能传输图片、视频等任意数据，非常方便。
2. **无状态**。这里的状态是指**通信过程的上下文信息**，而每次 http 请求都是独立、无关的，默认不需要保留状态信息。
3. **无连接** 无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间
4. **简单快速** 客户向服务器请求服务时，只需传送请求方法和路径。请求方法常用的有GET、HEAD、POST。每种方法规定了客户与服务器联系的类型不同。由于HTTP协议简单，使

得HTTP服务器的程序规模小，因而通信速度很快

缺点:

1. **无状态** 对于长链接的场景来说, 需要保存大量的上下文信息，以免传输大量重复的信息，那么这时候无状态就是 http 的缺点了。
2. **明文传输** 协议里的报文 是文本形式 这当然对于调试提供了便利，但同时也让 HTTP 的报文信息暴露给了外界，给攻击者也提供了便利。 `WIFI陷阱` 就是利用 HTTP 明文传输的缺点，诱导你连上热点，然后疯狂抓你所有的流量，从而拿到你的敏感信息
3. **队头阻塞问题 ** 当开始长链接. 同一时刻只能处理一个请求. 请求时间过长的情况下. 其他请求只能处于阻塞状态

HTTP1.1 如何解决 HTTP 的队头阻塞问题?

- **并发连接**
    - 对于一个域名允许分配多个长连接，那么相当于增加了任务队列，不至于一个队伍的任务阻塞其它所有任务。在RFC2616规定过客户端最多并发 2 个连接，不过事实上在现在的浏览器标准中，这个上限要多很多，Chrome 中是 6 个。
- **域名分片**
    - 一个域名不是可以并发 6 个长连接吗？那我就多分几个域名。

      比如 content1.sanyuan.com 、content2.sanyuan.com。

      这样一个 `sanyuan.com` 域名下可以分出非常多的二级域名，而它们都指向同样的一台服务器，能够并发的长连接数更多了，事实上也更好地解决了队头阻塞的问题。

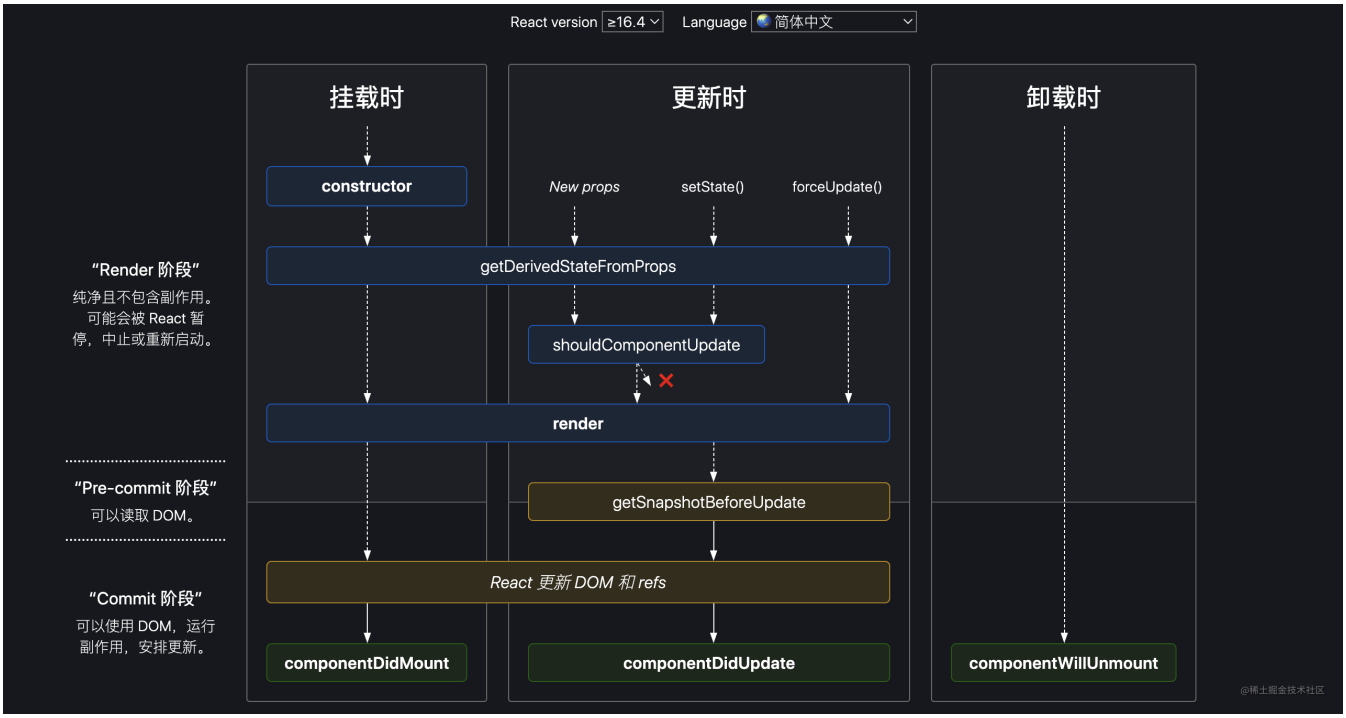
React

为什么要自定义事件机制?

- 抹平浏览器差异，实现更好的跨平台。
- 避免垃圾回收，React 引入事件池，在事件池中获取或释放事件对象，避免频繁地去创建和销毁。
- 方便事件统一管理和事务机制。

Class component

生命周期



挂载阶段: constructor() -> getDerivedStateFromProps() 返回一个对象更新state, 是否更新state -> render() -> 更新Dom树 -> componentDidMount()

更新阶段: getDerivedStateFromProps() -> shouldComponentUpdate() 是否更新这次. -> render() -> getSnapShotBeforeUpdate() 拿到更新前的Dom相关值 -> 更新Dom树 -> componentDidUpdate()

卸载阶段: componentWillUnmount()

hook

为什么不能在条件语句中写 hook

hook 在每次渲染时的查找是根据一个"全局"的下标对链表进行查找的，如果放在条件语句中使用，有一定几率会造成拿到的状态出现错乱.

HOC 和 hook 的区别

hook 复用**逻辑**. HOC复用**逻辑和视图**

useEffect 和 useLayoutEffect 区别

函数组件一次state更新的过程:

1. 某state发生了变化
2. React内部更新state
3. 处理更新组件的Dom节点
4. Dom更新渲染
5. Dom更新后 (Layout阶段后)

`useEffect` 在第 4 步之后执行，且是异步的，保证了不会阻塞浏览器进程。 `useLayoutEffect` 在第 3 步至第 4 步之间执行，且是同步代码，所以会阻塞后面代码的执行

useEffect 依赖为空数组与 componentDidMount 区别

在 `render` 执行之后， `componentDidMount` 会执行，如果在这个生命周期中再一次 `setState` ，会导致再次 `render` ，返回了新的值，浏览器只会渲染第二次 `render` 返回的值，这样可以避免闪屏。

但是 `useEffect` 是在真实的 DOM 渲染之后才会去执行，这会造成两次 `render` ，有可能会闪屏。(useEffect时commit完成后异步调用)

实际上 `useLayoutEffect` 会更接近 `componentDidMount` 的表现，它们都同步执行且会阻碍真实的 DOM 渲染的。 useLayoutEffect和componentDidMount调用时机一致，也是在layout阶段同步调用.

React.memo() 和 React.useMemo() 的区别

- memo是高阶组件. 浅比较props, 引用类型可以用第二个参数手动比价
- useMemo 返回一个缓存值. 只有依赖项发生变化才会重新执行.
- useCallback 根据依赖项缓存函数
    - 有很多时候，我们在 `useEffect` 中使用某个定义的外部函数，是要添加到 `deps` 数组中的，如果不用 `useCallback` 缓存，这个函数在每次重新渲染时都是一个完全新的函数，也就是引用地址发生了变化，这就会导致 `useEffect` 总会无意义的执行。

Webpack

webpack5 和 webpack4 的区别有哪些？

待更新

经历

虚拟DOM

- 虚拟 DOM 最大的优势在于抽象了原本的渲染过程，实现了跨平台的能力，而不仅仅局限于浏览器的 DOM，可以是安卓和 IOS 的原生组件，可以是近期很火热的小程序，也可以是各种 GUI。
- 实现了对 DOM 的集中化操作，在数据改变时先对虚拟 DOM 进行修改，再反映到真实的 DOM 中，用最小的代价来更新 DOM，提高效率(提升效率要想想是跟哪个阶段比提升了效率，别只记住了这一条)。

> 缺点

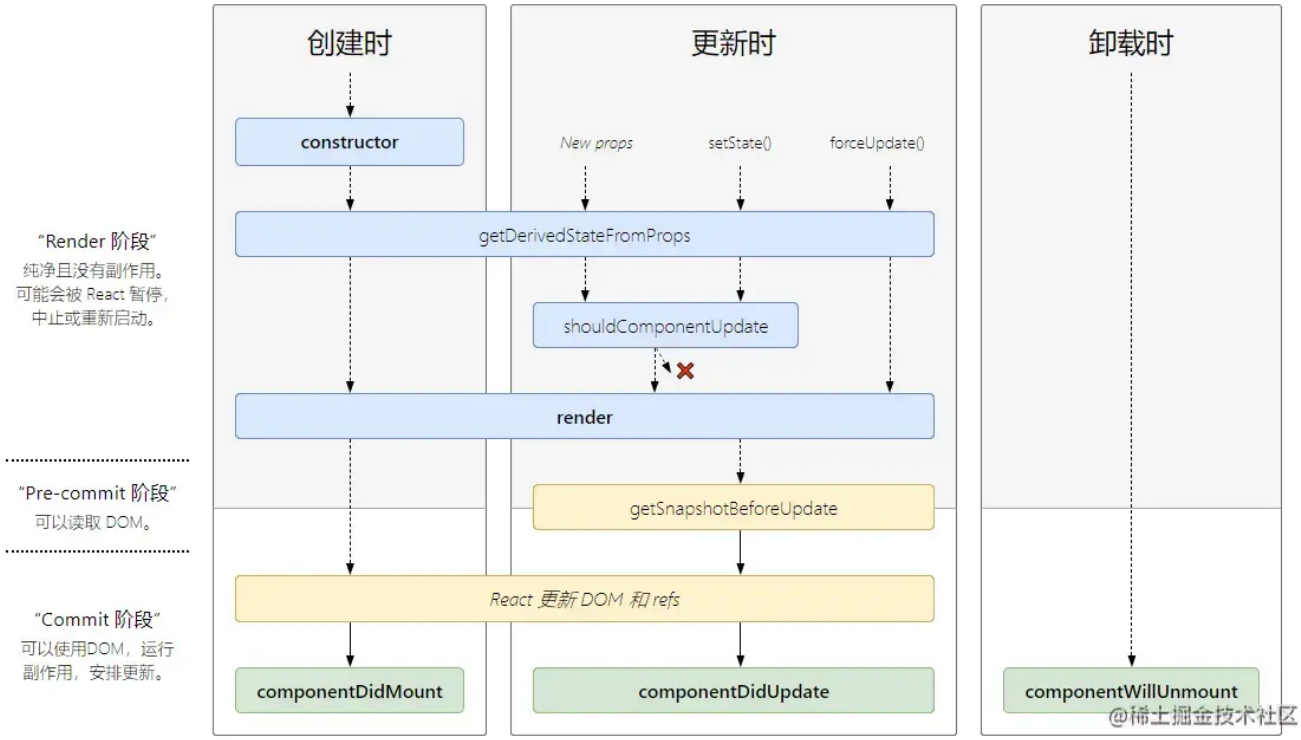
- 首次渲染大量 DOM 时，由于多了一层虚拟 DOM 的计算，会比 innerHTML 插入慢。(首屏慢)
- 虚拟 DOM 需要在内存中的维护一份 DOM 的副本 (空间上)
-

react使用优势?

1. 虚拟Dom
2. 组件化 将页面拆分成若干个组件，并且react一个组件同时耦合了css，js ,image，这种模式整个垫付了过去的传统模式
3. JSX语法
4. Props单向数据流

是个库. 项目使用需要引入react-router, 数据管理react-redux配合

react生命周期



react 怎么diff?

react的diff 策略

- **同级diff, 忽略跨层级的移动操作**
- **不同类型的组件生成不同的树**
- **同层级子节点可以使用key作为区分**

其中Diff分为两类：

1. 当 `newChild` 类型为 `object` 、 `number` 、 `string` ， 代表同级只有一个节点
2. 当 `newChild` 类型为 `Array` ， 同级有多个节点

同级的节点的diff分为: **单节点diff**: 先看上次的更新child是否存在.

markdown 复制代码

```
- 不存在  新建fiber节点
- 存在看是否能够复用
        - 先看key是否相同
                - key不相同  将该fiber标记为删除，后面可能还有兄弟fiber.
                - key相同  接下来看type是否相同
                        - type相同则复用
                        - type不相同  将该fiber及其兄弟fiber全部标记为删除
```

多节点diff:

> 虽然本次更新的JSX对象 newChildren为数组形式，但是和newChildren中每个组件进行比较的是current fiber，同级的Fiber节点是由sibling指针链接形成的单链表，即不支持双指针遍历。
>
> 在日常开发中，相较于 新增 和 删除 ， 更新 组件发生的频率更高。 所以 Diff 会优先判断当前节点是否属于 更新 。

Diff 分两轮遍历:

第一轮遍历:处理更新的节点 第二轮遍历: 处理剩下的不属于更新的节点

第一轮遍历:

1. `let newidx = 0` , 遍历 `newChildren` ; 将 `newChildren[i]` 与 `oldFilber` 进行比较. 如果可以复用, 继续遍历
2. `newidx++` ; 继续将 `newChildren[i]` 与 `oldFilber.sibling` 进行比较 . 如果可以复用, 继续遍历
3. 如果不可以复用, 分两种:
    1. `key` 不同 **直接第一轮遍历结束**
    2. `key` 相同 `type` 不同. 将oldFilber标记删除, 继续遍历
4. 如果 `newChildren` 遍历完 (`newIdx === newChildren.length - 1`) 或者 `oldFiber` 遍历完 `oldFiber === bull` , 跳出遍历，**第一轮遍历结束。**

第一轮遍历的结果会有几种情况:

1. `newChildren` 与 `oldFiber` 都遍历完了 --- 说明这次只是更新节点， `Diff` 结束
2. `newChilderen` 遍历完了，`oldFilber` 没有遍历完 --- 说明这次删除节点, 只需将剩下 `oldFilber` 标记 `deletion`
3. `newChilderen` 没有遍历完，`oldFilber` 遍历完了 --- 说明这次新增节点, 只需将剩下 `newChilderen` 遍历生成 `filber` 标记为新增
4. `newChilderen` 和 `oldFilber` 都没有遍历完, --- 说明这次更新中改变了位置

第二轮遍历:

> 没遍历完的节点

由于有节点改变了位置，所以不能再用位置索引 `i` 对比前后的节点，那么如何才能将同一个节点在两次更新中对应上呢?

我们需要使用 `key` 。

为了快速的找到 `key` 对应的 `oldFiber` ， 我们将所有还未处理的 `oldFiber` 存入以 `key` 为key， `oldFiber` 为value的 `Map` 中。

```javascript 复制代码
const existingChildren = mapRemainingChildren(returnFiber, oldFiber);
```

接下来遍历剩余的 `newChildren` ， 通过 `newChildren[i].key` 就能在 `existingChildren` 中找到 `key` 相同的 `oldFiber` 。

- 找到相同key
  - type是否相同 可复用
    - 可复用 existingChildren.delete(newFiber.key)
  - 不可复用 新建fiber
- 没找到新建fiber

剩余existingChildren如果还有 遍历筛入deletions

改变了位置的情况

react性能优化? 项目上的性能优化?

React(代码层面):

- 组件: React.useMemo; PureComponent, shouldComponentUpdate
- 函数: useCallback. useMemo;
- 避免内联函数.
- React.lazy与 React.Suspense
- 减少渲染(重排和重绘)
- 防抖和节流（resize，scroll，input）。

构建上的性能优化:

- **CDN加速**: **output.publicPath** 静态资源配置cdn
- **压缩css** plugin: `optimize-css-assets-webpack-plugin` `cssnano`
- **压缩html**: `HtmlWebpackPlugin.minify`
- **tree shaking 清除无用的css, js**:

    - css: glob-all@3.2.1 `purify-css@1.2.5` `purifycss-webpack@0.7.0`
    - js: `optimization.usedExports` : true 且只有mode: 'production'才生效. 消除副作用: package.json中的 sideEffects: ['*.less*', '.css', '@babel/polyfill'] // 免除消除
- **代码分割 optimization.splitChunks **: 分离公共文件.
- **开启 gzip 压缩** webpack 中使用 `compression-webpack-plugin` ，node 作为服务器也要开启，使用 `compression` 。

其它:

- 使用 http2。因为解析速度快，头部压缩，多路复用，服务器推送静态资源。
- 使用服务端渲染。
- 图片压缩。
- 使用 http 缓存，比如服务端的响应中添加 `Cache-Control / Expires` 。

http缓存

react组件间通信

参考答案:

以下6种方法是react组件间通信的方式:

- 父组件向子组件通讯: 父组件可以向子组件通过传 props 的方式，向子组件进行通讯
- 子组件向父组件通讯: props+回调的方式,父组件向子组件传递props进行通讯，此props为作用域为父组件自身的函数，子组件调用该函数，将子组件想要传递的信息，作为参数，传递到父组件的作用域中
- 兄弟组件通信: 找到这两个兄弟节点共同的父节点,结合上面两种方式由父节点转发信息进行通信
- 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，例如当前认证的用户、主题或首选语言，对于跨越多层的全局数据通过Context通信再适合不过
- 发布订阅模式: 发布者发布事件，订阅者监听事件并做出反应,我们可以通过引入event模块进行通信
- 全局状态管理工具: 借助Redux或者Mobx等全局状态管理工具进行通信,这种工具会维护一个全局状态中心Store,并根据不同的事件产生新的状态

webpack 压缩js的插件?

多线程压缩js: webpack-parallel-uglify-plugin,

- `uglifyjs-webpack-plugin`： 不支持 ES6 压缩 (Webpack4 以前)
- `terser-webpack-plugin`：支持压缩 ES6 (Webpack4)

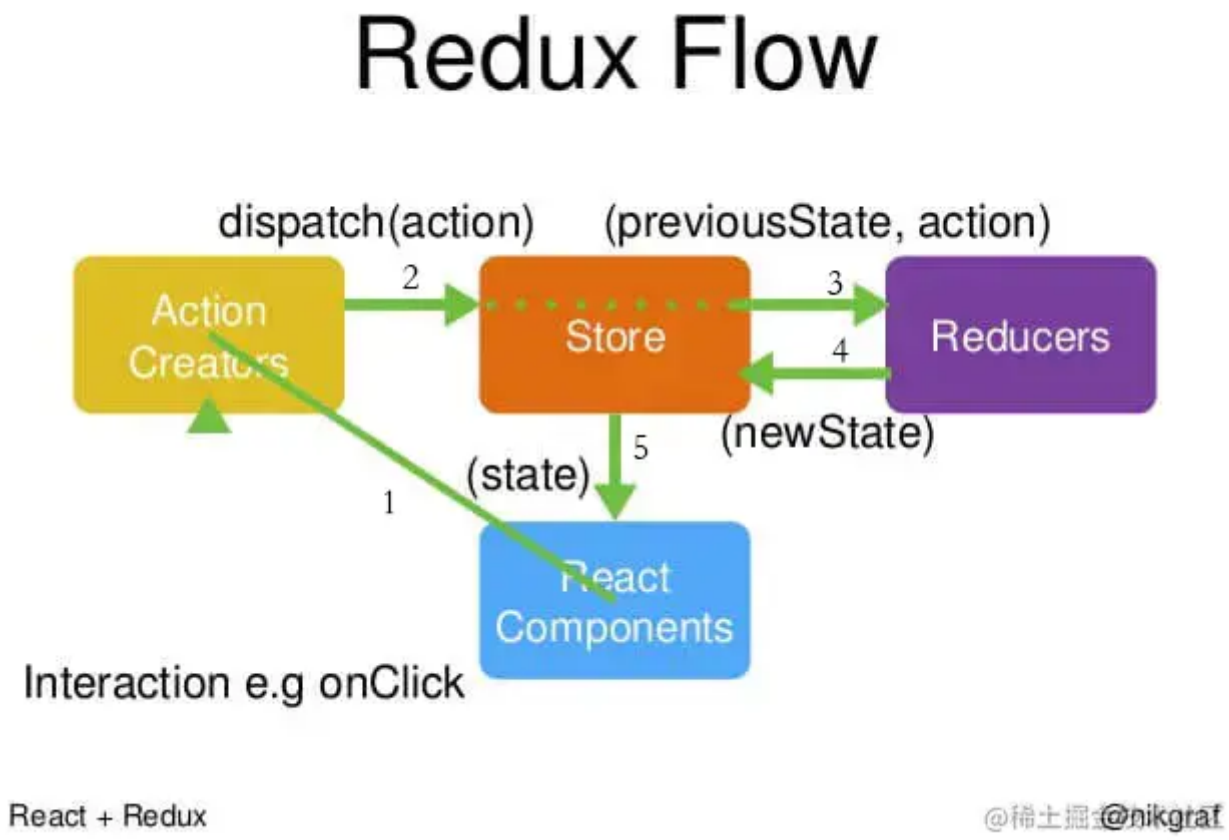
hook与class组件的区别?

- hook写法更简洁. 比class更 声明式
- hook使得函数组件更容易复用. 比class更 函数式
- 复杂组件中 class起初简单. 后面逐渐被状态逻辑和副作用充斥. 每个生命周期包含不相关的逻辑. 难以拆分. 而hook,可以将组件相互关联的部分拆分成更小的函数, 例如请求数据, 设置订阅等. 而非按生命周期来划分.
- hook可以使你在无需改变组件结构的情况下复用状态逻辑. Class组件复用状态逻辑需要使用高级组件嵌套,或者provider. render props

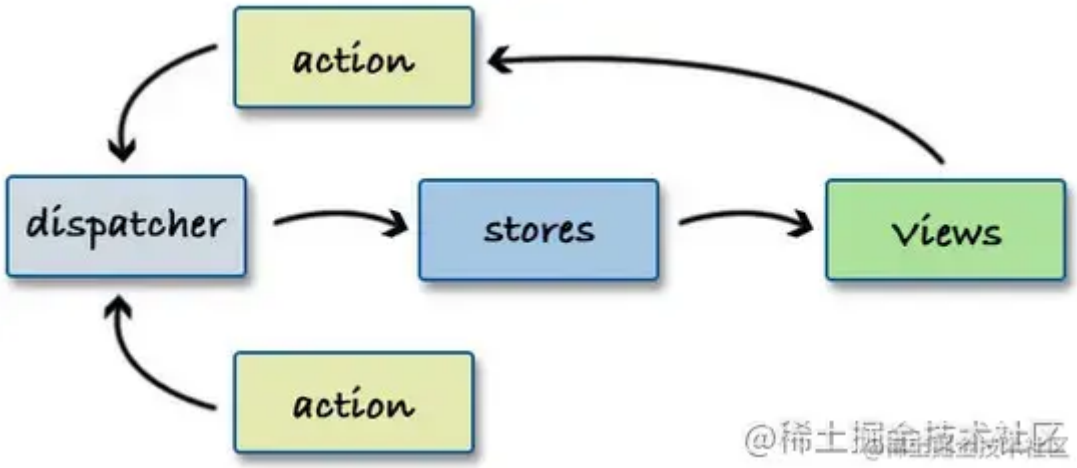
使用typescript的好处?

- 增加了代码的可读性和可维护性
  - 类型系统实际上是最好的文档
  - 可以在编译阶段就发现大部分错误
  - 增强了编辑器和 IDE 的功能，包括代码补全、接口提示、跳转到定义、重构等
  - 更好的协作
- 静态类型检查

redux使用流程?



首先由view dispatch拦截action，然后执行对应reducer并更新到store中，最终views会根据store数据的改变执行界面的刷新渲染操作。



- **单一数据源**
- **State是只读的**
- **应用状态的改变通过纯函数来完成**

redux异步处理 redux-thunk

浅拷贝的方法, 且解构符与浅拷贝的区别?

解构赋值是浅拷贝一层, 属性如果是对象, 是对象引用

```js
// 对象
var obj1 = {a: 1, b: 2}
// Object.assign
 var obj2 = Object.assign({}, obj1)
// 解构赋值

// 数组
// slice()
var arr = arr.slice()
var arr = [].concat(arr)
// 遍历
// 展开符
```

react中setState与Vue的赋值区别?

两者对数据的处理不一样

- vue 是数据的劫持和通知订阅的方式，所以可以立即响应数据的变化。 值的改变却是同步的, dom的更新是异步的
- react数据更新setState是异步的

小程序经验过期 分包处理

v-if v-show 是否会导致重排重绘?

（1）两者都会导致页面的重绘和重排，但v-show只是改变dom的css，而v-if控制的是添加和删除dom，所以v-if在重绘重排前还进行了添加或删除dom元素的操作。 （2）需要多次切换某个元素的显示或隐藏时使用v-show （3）某个元素在渲染后就一直存在或隐藏时使用v-if （4）opacity也可以隐藏元素，但它本身的作用并非用来隐藏元素而是设置元素的透明度，并且opacity为0时，该dom同样占用着空间。

hash 和 history的使用场景?

Hash:

原理:

- 早期的前端路由的实现就是基于 `location.hash` 来实现的， `location.hash` 的值就是URL中#后面的内容 其实现原理就是监听#后面的内容来发起Ajax请求来进行局部更新，而不需要刷新整个页面。

- 使用 `hashchange` 事件来监听 URL 的变化，以下这几种情况改变 URL 都会触发 hashchange 事件：浏览器前进后退改变 URL、a标签改变 URL、window.location改变 URL。

优点:

- hash值的改变，都会在浏览器的访问历史中增加一个记录，所以可以通过浏览器的回退、前进按钮控制hash的切换 会覆盖锚点定位元素的功能

History模式

- history 提供了 pushState 和 replaceState 两个方法来记录路由状态，这两个方法改变 URL 不会引起页面刷新

- history 提供类似 hashchange 事件的 popstate 事件，但 popstate 事件有些不同：通过浏览器前进后退改变 URL 时会触发 popstate 事件，通过pushState/replaceState或a标签改变 URL 不会触发 popstate 事件。好在我们可以拦截 pushState/replaceState的调用和a标签的点击事件来检测 URL 变化，所以监听 URL 变化可以实现，只是没有 hashchange 那么方便。

- pushState(state, title, url) 和 replaceState(state, title, url)都可以接受三个相同的参数。

两种不同使用场景

- 从上文可见，hash模式下url会带有#，当你希望url更优雅时，可以使用history模式。
- 当使用history模式时，需要注意在服务端增加一个覆盖所有情况的候选资源：如果 URL 匹配不到任何静态资源，则应该返回同一个 index.html 页面，这个页面就是你 app 依赖的页面。
- **当需要兼容低版本的浏览器时，建议使用hash模式。**
- **当需要添加任意类型数据到记录时，可以使用history模式。**

SEO meta的知识复习

```html 复制代码
<meta name='description' content={`${description}`} />
<meta name='keywords' content={`${keywords}`} />
```

参考： www.seozac.com/seo-tips/js...

- **搜索引擎怎样处理JS?**
  - JS造成SEO问题的症结在于，搜索引擎不一定执行JS脚本。
  - Google遇到页面JS时，会在有计算资源、且页面有比较高价值时，尝试执行脚本、渲染页面。百度则基本上不执行JS脚本。所以做中文网站，使用JS上就更要谨慎。
- **重要链接不要用JS**
  - 搜索引擎爬行、抓取页面是靠跟踪链接的。如果重要链接需要运行JS脚本才能调用或解析出来，那搜索引擎就可能无法跟踪。
- **想被收录的内容不要用JS调用**
  - 页面上的文字内容，凡是想被收录的，不要用JS调用，包括文章正文，产品说明，产品图片，评论等。
- **慎用懒加载、瀑布流**
  - 图片懒加载，甚至文字内容懒加载，是现在网站经常使用的方法，在一定程度上有利于提高页面速度。但要注意，用JS实现懒加载时，是否需要用户互动才能加载，比如点击“更多”链接，或者向下拉页面，搜索引擎蜘蛛是不会做这些动作的，不会点击按钮，也不会下拉页面，所以就可能看不到懒加载后的内容。无论懒加载的是更多本页内容，还是更多其它页面列表，都可能造成爬行、索引问题。
- 注意速度
  - 除了用户体验，页面速度也是搜索排名的重要因素。

解决方法：

- **服务器端渲染**
  - 但服务器性能一定是超过用户设备的，再加上缓存等方法，总体上是会比浏览器执行JS、渲染页面快很多的。
- 怎样检查JS是否造成SEO问题
  - 浏览器禁用JS
  - 检查页面的快照

前端安全漏洞

XSS攻击: 一种 代码注入攻击 ， 通过恶意注入脚本在浏览器运行，然后盗取用户信

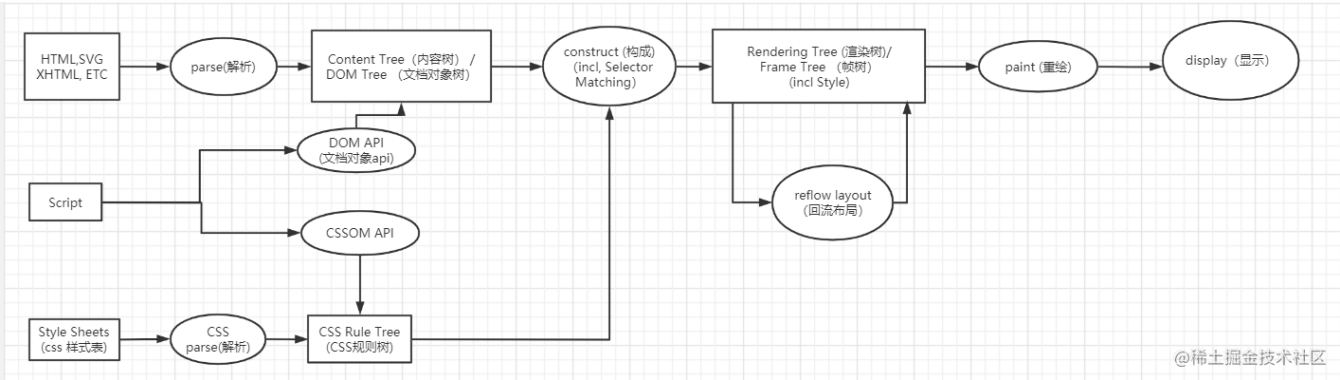
- cookie配置:
  - httpOnly : 不允许js修改
  - secure: https环境
- 过滤转义输入输出 输入校验
- 综合策略 X-Content-Security-Policy 响应头（meta配置）

CSRF攻击: 跨站请求伪造攻击 ， *主要就是利用用户的登录状态发起跨站*

- 登录token过期机制
- 检测http referer 是否是白名单
- 关键接口使用验证码

浏览器渲染过程

解析HTML->构建DOM -> 构建CSSOM -> 构建**渲染**树-> 布局(回流->重绘)-> 绘制。



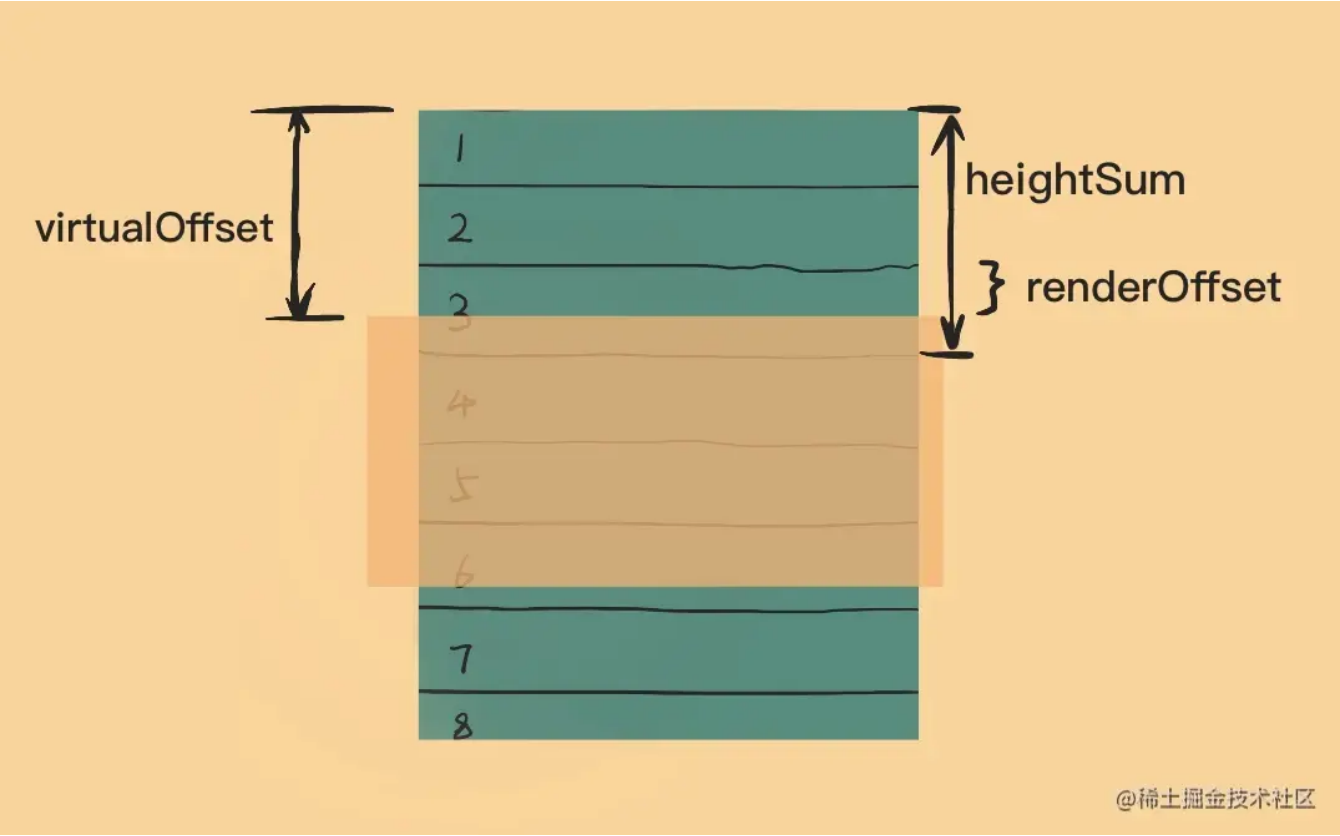
浏览器是单线程吗

浏览器内核是多线程

- GUI 渲染线程
- JavaScript引擎线程

- 定时触发器线程
- 事件触发线程
- 异步http请求线程

虚拟列表 滚动位置的实现



滚动条配置:

10万条; 滚动条比例过小. 给定最小高度. 按照比例来赋值

- 视口能渲染几个列表元素?
  - 需要我们给每一个列表元素设置一个高度。通过累加高度计算找到第一个加完它的高度后总高度超出视口高度的列表元素。
- 怎么知道该渲染哪几个元素?
  - 累加的元素高度 > 总偏移量 的时候返回元素序号
  - 视角的第一个元素的偏移量 = 总偏移量 - 累加元素的高度和 + 视角的第一元素的高度
- 列表元素咋渲染成我想要的?
  - 渲染列表视角的第一个元素 translateY

性能优化:

- 事件节流
- 列表缓存(eg: 上下多渲染10条) 滚动量小的时候不用重新渲染列表

跨域名通信的方法

同源页面之间通信

- broadcast channel
- service worker
- LocalStorage
- indexedDB
- shared worker

非同源页面通信

- iframe
- postMessage

设计模式

参考： www.jianshu.com/p/993027963...

- 单例模式
- 策略模式 （针对不同规则有不同的返回、处理）
- 代理模式 （防抖）
- 发布订阅（观察者）模式
- 组合模式 （有相同的方法， 形成一个命令统一执行）

闭包的好处, 优点

- 闭包就是能够读取其他函数内部变量的函数
- 由于在javascript中，只有函数内部的子函数才能读取局部变量，所以说，**闭包可以简单理解成"定义在一个函数内部的函数。**
- 在本质上，闭包是将函数内部和函数外部连接起来的桥梁 。

最大用处有两个，一个是前面提到的可以读取函数内部的变量，另一个就是让这些变量的值始终保持在内存中，不会在f1调用后被自动清除 。

好处:

1. 希望一个变量长期保存内存中;
2. 避免全局变量污染;
3. 私有成员的存在。

缺点:

1. 常驻内存，增加内存使用量;
2. 使用不当造成内存泄漏。

回流布局(重排)

Layout(回流):根据生成的渲染树，进行回流(Layout)，得到节点的几何信息（位置，大小）

css3硬件加速（GPU加速） 使用css3硬件加速，可以让transform、opacity、filters这些动画不会引起回流重绘 。但是对于动画的其它属性，比如background-color这些，还是会引起回流重绘的，不过它还是可以提升这些动画的性能。

request 缓存

SWR

Fiber

`React` 内部实现的一套状态更新机制。支持任务不同 `优先级` ，可中断与恢复，并且恢复后可以复用之前的 `中间状态` 。

其中每个任务更新单元为 `React Element` 对应的 `Fiber节点` 。

1. 为什么需要fiber
    - 对于大型项目，组件树会很大，这个时候递归遍历的 成本就会很高，会造成主线程被持续占用，结果就是 主线程上的布局、动画等周期性任务就无法立即得到 处理，造成视觉上的卡顿，影响用户体验。
2. 任务分解的意义

    - 解决上面的问题
3. 增量渲染（把渲染任务拆分成块，匀到多帧）
4. 更新时能够暂停，终止，复用渲染任务
5. 给不同类型的更新赋予优先级
6. 并发方面新的基础能力
7. 更流畅

fiber是指组件上将要完成或者已经完成的任务，每个组 件可以一个或者多个。

- 第一阶段: 找出更新列表, 可中断.
- 第二阶段: 提交到Dom, 状态不可中断, requestIdleCallback() React自己实现这个 scheduler `Scheduler` 将需要被执行的回调函数作为 `MessageChannel` 的回调执行。如果当前宿主环境不支持 `MessageChannel` ，则使用 `setTimeout` 。

React Fiber 是对核心算法的一次重新实现 Fiber reconciler 从 v16.x 开始底层使用 Fiber reconciler 替换 stack reconciler. 已知： stack reconciler 处理大状态时由于计算和组件树遍历的消耗容易出现渲染线程挂起，进而页面掉帧。（根本原因是渲染/更新过程一旦开始无法中断，持续占用主线程，主线程忙于执行 JS）

求： 建立一种能解决主线程占用问题，且具有长远意义的机制 解： 把渲染/更新过程拆分为小块任务，通过合理的调度机制来控制时间（更细粒度、更强的控制力）

子问题： 1.拆什么？什么不能拆？ 把渲染/更新过程分为 2 个阶段（diff + patch）： diff ~~render/reconciliation (对比 prevInstance 和 nextInstance 的状态，找出差异及其对应的 DOM change。) patch~~ commit (把本次更新中的所有 DOM change 应用到 DOM 树，是一连串的 DOM 操作。) render/reconciliation 阶段的工作（diff）可以拆分，commit 阶段的工作（patch）不可拆分.

2.怎么拆？ Fiber 的拆分单位是 fiber（fiber tree 上的一个节点），实际上就是按虚拟 DOM 节点拆，因为 fiber tree 是根据 vDOM tree 构造出来的，树结构一模一样，只是节点携带的信息有差异。

3.如何调度任务？ 分 2 部分： 工作循环 优先级机制 工作循环是基本的任务调度机制，工作循环中每次处理一个任务（工作单元），处理完毕有一次喘息的机会，此时通过 shouldYield 函数（idleDeadline.timeRemaining()）判读时间是否用完，用完则把时间还给主线程等待下次 requestIdleCallback 的唤起，否则继续执行任务。 优先级机制用来处理突发事件与优化次序。 有如下策略： 到 commit 阶段了，提高优先级 高优任务做一半出错了，给降一下优先级

抽空关注一下低优任务，别给饿死了 如果对应 DOM 节点此刻不可见，给降到最低优先级 是工作循环的辅助机制。

4.如何中断/断点恢复？ 中断：检查当前正在处理的工作单元，保存当前成果（firstEffect, lastEffect），修改 tag 标记一下，迅速收尾并再开一个 requestIdleCallback，下次有机会再做 断点恢复：下次再处理到该工作单元时，看 tag 是被打断的任务，接着做未完成的部分或者重做 自然中断（时间耗尽），或优先级中断（高优任务中断），原理相同。

5.如何收集任务结果？ 每个节点更新结束时向上归并 effect list 来收集任务结果，reconciliation 结束后，根节点的 effect list 里记录了包括 DOM change 在内的所有 side effect。

requestIdleCallback 让开发者在主事件循环中执行后台或低优先级的任务,不会对动画和用户交互等关键事件产生影响。

fiber 架构：

- 循环条件：利用 requestIdeCallback 空闲时间递减.
- 遍历过程：利用链表，找孩子找兄弟找父亲

webpack兼容性处理

Css:

配置postcss-loader , 并且在package.json中browserslist配置要兼容的浏览器版本

```
js 复制代码
/*
    postcss-loader：css兼容性处理：postcss --> 需要安装：postcss-loader postcss-preset-env
    postcss需要通过package.json中browserslist里面的配置加载指定的css兼容性样式
    在package.json中定义browserslist：
    "browserslist": {
      // 开发环境 --> 设置node环境变量：process.env.NODE_ENV = development
      "development": [ // 只需要可以运行即可
        "last 1 chrome version",
        "last 1 firefox version",
        "last 1 safari version"
      ],
      // 生产环境。默认是生产环境
      "production": [ // 需要满足绝大多数浏览器的兼容
        ">0.2%",
        "not dead",
        "not op_mini all"
```

```
      ]
    },
  */
  {
    loader: 'postcss-loader',
    options: {
      ident: 'postcss', // 基本写法
      plugins: () => [
        // postcss的插件
        require('postcss-preset-env')(),
      ],
    },
  },
```

Js:

babel-loader

js 复制代码

```
/*
      js兼容性处理：需要下载 babel-loader @babel/core @babel/core: babel功能函数
        1. 基本js兼容性处理 --> @babel/preset-env
          问题：只能转换基本语法，如promise高级语法不能转换
        2. 全部js兼容性处理 --> @babel/polyfill
          问题：只要解决部分兼容性问题，但是将所有兼容性代码全部引入，体积太大了
        3. 需要做兼容性处理的就做：按需加载  --> core-js
      */
      {
        // 第三种方式：按需加载
        test: /\.js$/,
        exclude: /node_modules/,
        loader: 'babel-loader',
        options: {
          // 预设：指示babel做怎样的兼容性处理
          presets: [
            '@babel/preset-env', // 基本预设
            {
              useBuiltIns: 'usage', //按需加载
              corejs: { version: 3 }, // 指定core-js版本
              targets: { // 指定兼容到什么版本的浏览器
                chrome: '60',
                firefox: '50',
                ie: '9',
                safari: '10',
                edge: '17'
              },
            },
```

```
          ],
        },
      },
```

Node 常用模块

项目难点

1. 项目中抽屉点击任何地方收回
2. [虚拟列表](#虚拟列表 滚动位置的实现)
3. 物理机升级node版本,启动报错. 本地和线上不一致. 从而推动项目docker部署, 从而保证本地和线上环境一致
4. 项目是服务端渲染. 起了一个Socket.io
5. 第三方平台登录 postMessage, 父窗口打开第三方登录，跳转到不同域名。使用 window.opener.postMessage()
6. 跨页面交流 localStorage

数组扁平化

js 复制代码

```
function flat(arr) {
    var res = [];
    arr.forEach((item, i) => {
      if (Array.isArray(item)) {
        res = res.concat(flat(item));
      } else {
        res.push(item);
      }
    });
    return res;
  }
// 进阶控制扁平化层数
function flat(arr, level = 1) {
    var res = [];
    arr.forEach((item, i) => {
      if (Array.isArray(item)) {
        if(level > 0) {
          res = res.concat(flat(item, level - 1));
        }else {
          res.push(item)
        }
      } else {
        res.push(item);
```

```
      }
    });
    return res;
  }
```

浏览器缓存

缓存过程: 浏览器发去HTTP请求 -> 浏览器缓存是否有缓存标识和缓存结果且是否有效 -> 有缓存标识发送给服务器返回请求结果(有标识无效/没标识-> 200+新结果)(有标识且有效 -> 304 继续使用缓存) -> 浏览器存储请求结果到缓存中.

Expires(HTTP/1.0)值为服务器该缓存结果到期时间. 优先级低于Cache-Control(HTTP/1.1) . 原因是Expires可能存在时区误差.

If-Modified-Since值为Last-Modified(该资源最后修改时间), 客户端再次发送请求时携带该值与服务器该资源最后修改时间做对比. 优先级低于If-None-Match值为Etag.

If-None-Match值为Etag(服务器生成该资源的唯一标识), 客户端再次发送请求时携带该值与服务器Etag做对比.

强制缓存优先于协商缓存进行. 若强制缓存(Expires和**Cache-Control**)生效则直接使用缓存，若不生效则进行协商缓存(Last-Modified / If-Modified-Since和**Etag / If-None-Match**)，**协商缓存由服务器决定是否使用缓存**，若协商缓存失效，那么代表该请求的缓存失效，重新获取请求结果，再存入浏览器缓存中；生效则返回304，继续使用缓存.

Hook中定时器的使用

zh-hans.reactjs.org/docs/hooks-...

jsx 复制代码

```jsx
function Counter() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const id = setInterval(() => {
      setCount(c => c + 1); // ✅ 在这不依赖于外部的 `count` 变量
    }, 1000);
    return () => clearInterval(id);
  }, []); // ✅ 我们的 effect 不使用组件作用域中的任何变量
```

```
  return <h1>{count}</h1>;
}
```

有依赖的情况

jsx 复制代码

```jsx
function Example(props) {
  // 把最新的 props 保存在一个 ref 中
  const latestProps = useRef(props);
  useEffect(() => {
    latestProps.current = props;
  });

  useEffect(() => {
    function tick() {
      // 在任何时候读取最新的 props
      console.log(latestProps.current);
    }

    const id = setInterval(tick, 1000);
    return () => clearInterval(id);
  }, []); // 这个 effect 从不会重新执行
}
```

jsx 复制代码

```jsx
import { useEffect, useRef } from 'react';

const useInterval = (callback: Function, delay?: number | null) => {
  const savedCallback = useRef<Function>(() => {});

  useEffect(() => {
    savedCallback.current = callback;
  });

  useEffect(() => {
    if (delay !== null) {
      const interval = setInterval(() => savedCallback.current(), delay || 0);
      return () => clearInterval(interval);
    }

    return undefined;
  }, [delay]);
};

export default useInterval;
```

URL访问过程

域名DNS解析(获取服务器IP地址) ==> 与服务器建立TCP连接(三次握手) ==> 服务器响应请求 ==> 客户端接收数据开始解析渲染

> 客户端下载、解析、渲染显示页面

- a. 如果是Gzip包，则先解压为HTML
- b. 解析HTML的头部代码，下载头部代码中的样式资源文件或脚本资源文件
- c. 解析HTML代码和样式文件代码，构建HTML的DOM树以及与CSS相关的CSSOM树
- d. 通过遍历DOM树和CSSOM树，浏览器依次计算每个节点的大小、坐标、颜色等样式，构造渲染树
- e. 根据渲染树完成绘制过程

websocket原理

Websocket是一个**持久化**的协议

websocket约定了一个通信的规范，通过一个握手的机制，客户端和服务器之间能建立一个类似tcp的连接，从而方便它们之间的通信。在websocket出现之前，web交互一般是基于http协议的短连接或者长连接。websocket是一种全新的协议，不属于http无状态协议，协议名为"ws"，这意味着一个websocket连接地址会是这样的写法：ws://**。websocket协议本质上是一个基于tcp的协议。

www.infoq.cn/article/ujw...

当客户端要和服务端建立 WebSocket 连接时，在客户端和服务器的握手过程中，客户端首先会向服务端发送一个 HTTP 请求，包含一个 Upgrade 请求头来告知服务端客户端想要建立一个 WebSocket 连接。

BFC

BFC的原理（渲染规则）

1. BFC元素垂直方向的边距会发生重叠。**属于不同BFC外边距不会发生重叠**
2. BFC的区域**不会与浮动元素的布局重叠。**
3. BFC元素是一**个独立的容器，外面的元素不会影响里面的元素。里面的元素也不会影响外面的元素。**

4. **计算BFC高度的时候，浮动元素也会参与计算**(清除浮动)

如何创建BFC

1. **overflow不为visible**;
2. float的值不为none;
3. position的值不为static或relative;
4. **display属性为inline-blocks,table,table-cell,table-caption,flex,inline-flex**;

使用情况

1. BFC内的外边距不与外部的外边距发生重叠。(非垂直)
2. BFC元素不会与浮动元素发生重叠(在没BFC时, 浮动元素重叠, 但是文本信息会被浮动元素所覆盖)。**BFC的话. 兄弟元素浮动不重叠**
3. 子元素都浮动. **BFC的话. 计算父元素高度的时候，浮动元素也会参与计算. (清除浮动)**

数组去重/排序

```
js 复制代码

// 1. [...new Set(arr)]
// 2. 遍历 map
// 3. 使用栈结构
// 4. 排序后，双指针遍历 使用两个指针，右指针始终往右移动，
    //  如果右指针指向的值等于左指针指向的值，左指针不动。
    //  如果右指针指向的值不等于左指针指向的值，那么左指针往右移一步，然后再把右指针指向的值赋给左指针。
```

- 冒泡排序 （挨个对比，交换位置）
- 插入排序 （定一个排好序的数组， 对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。）
- 希尔排序 （插入排序的升级版）
- 快速排序 （定一个标志位）
- 堆排序 （使用大顶堆 **arr[i] >= arr[2i+1] && arr[i] >= arr[2i+2] ** ）
- 归并排序 （先分在合）

HTTP 无状态

特点： 无连接 、 无状态 、 灵活 、 简单快速

- **无连接**：每一次请求都要连接一次，请求结束就会断掉，不会保持连接
- **无状态**：每一次请求都是独立的，请求结束不会记录连接的任何信息(**提起裤子就不认人的意思**)，减少了网络开销，这 是优点也是缺点
- **灵活**：通过http协议中头部的 `Content-Type` 标记，可以传输任意数据类型的数据对象(文本、图片、视频等等)，非常灵活
- **简单快速**：发送请求访问某个资源时，只需传送请求方法和URL就可以了，使用简单，正由于http协议简单，使得http服务器的程序规模小，因而通信速度很快

缺点： 无状态 、 不安全 、 明文传输 、 队头阻塞

- **无状态**：请求不会记录任何连接信息，没有记忆，就无法区分多个请求发起者身份是不是同一个客户端的，意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大
- **不安全**： 明文传输 可能被窃听不安全，缺少 身份认证 也可能遭遇伪装，还有缺少 报文完整性验证 可能遭到篡改
- **明文传输**：报文(header部分)使用的是明文，直接将信息暴露给了外界， `WIFI陷阱` 就是复用明文传输的特点，诱导你连上热点，然后疯狂抓取你的流量，从而拿到你的敏感信息
- **队头阻塞**：开启 长连接 (下面有讲)时，只建立一个TCP连接，同一时刻只能处理一个请求，那么当请求耗时过长时，其他请求就只能阻塞状态(如何解决下面有讲)

webpack的优化

- 优化打包速度(构建速度)
    - 提升效率
        - **给loader加include**: 缩小查找范围
        - **添加resolve.modules**: 指定查找第三方库的位置 eg: modules: [path.resolve(__dirname, "./node_modules")],
        - **externals添加cdn隐射**. cdn资源. eg: 把react扔到cdn. html模板引入cdn. 项目开发正常使用import
        - **合理使用别名 resolve.alias**: 减少查找过程,直接使用打包好的代码. 节省解析时间. eg: alias: { react: path.resolve(

          __dirname,

"./node_modules/react/umd/react.production.min.js"

) }

- **speed-measure-webpack-plugin**@1.3.3: 可以测量各个插件和 loader 所花费的时间 分析出 Webpack 打包过程中 Loader 和 Plugin 的耗时，有助于找到构建过程中的性能瓶颈。简称 SMP

- **DllPlugin**: 采用webpack的 DllPlugin 和 DllReferencePlugin 引入dll，**让一些基本不会改动的代码先打包成静态资源,避免 反复编译浪费时间**

- 优化构建速度

  - **HardSourceWebpackPlugin**: 提供中间缓存的作用 第二次构建时间会有较大的节省
  - **thread-loader** 多进程打包

- 优化压缩速度

  - **webpack-parallel-uglify-plugin** 多线程压缩js

- 优化使用体验

- 优化输出质量(生产环境的质量)

  - 优化要发布到线上的代码，减少用户能感知到的加载时间
  - **提升代码性能**，性能好，执行就快
    - **CDN加速**: **output.publicPath** 静态资源配置cdn
    - **压缩css** plugin: `optimize-css-assets-webpack-plugin` `cssnano`
    - **压缩html**: `HtmlWebpackPlugin.minify`
    - **tree shaking 清除无用的css, js**:
      - css: glob-all@3.2.1 `purify-css@1.2.5` `purifycss-webpack@0.7.0`
      - js: `optimization.usedExports` : true 且只有mode: 'production'才生效. 消除副作用: package.json中的 sideEffects: ['*.less*', '.css', '@babel/polyfill'] // 免除消除
    - **代码分割 optimization.splitChunks **: 分离公共文件.
    - **通过配置 optimization.concatenateModules**: true` ： 开启 Scope Hoisting -- 通过 ES6 语法的静态分析， 分析出模块之间的依赖关系， 尽可能地把模块放到同一个函数中。

有一堆请求， 需求是串行处理

- async await
- Generator

useMemo 和 useCallback的实现

react.iamkasong.com/hooks/useme...

```
// 根据相关依赖项 判断依赖项是否改变。 无改变返回上次计算的结果
// 分mount阶段 update阶段
// memo callback的区别 memo是将回调函数的执行结果保存 callback将回调函数保存。 保存在当前hook.memoizedSta

function mountMemo<T>(
  nextCreate: () => T,
  deps: Array<mixed> | void | null,
): T {
  // 创建并返回当前hook
  const hook = mountWorkInProgressHook();
  const nextDeps = deps === undefined ? null : deps;
  // 计算value
  const nextValue = nextCreate();
  // 将value与deps保存在hook.memoizedState
  hook.memoizedState = [nextValue, nextDeps];
  return nextValue;
}

function mountCallback<T>(callback: T, deps: Array<mixed> | void | null): T {
  // 创建并返回当前hook
  const hook = mountWorkInProgressHook();
  const nextDeps = deps === undefined ? null : deps;
  // 将value与deps保存在hook.memoizedState
  hook.memoizedState = [callback, nextDeps];
  return callback;
}



function updateMemo<T>(
  nextCreate: () => T,
  deps: Array<mixed> | void | null,
): T {
  // 返回当前hook
  const hook = updateWorkInProgressHook();
  const nextDeps = deps === undefined ? null : deps;
  const prevState = hook.memoizedState;
```

js 复制代码

```
  if (prevState !== null) {
    if (nextDeps !== null) {
      const prevDeps: Array<mixed> | null = prevState[1];
      // 判断update前后value是否变化
      if (areHookInputsEqual(nextDeps, prevDeps)) {
        // 未变化
        return prevState[0];
      }
    }
  }
  // 变化，重新计算value
  const nextValue = nextCreate();
  hook.memoizedState = [nextValue, nextDeps];
  return nextValue;
}

function updateCallback<T>(callback: T, deps: Array<mixed> | void | null): T {
  // 返回当前hook
  const hook = updateWorkInProgressHook();
  const nextDeps = deps === undefined ? null : deps;
  const prevState = hook.memoizedState;

  if (prevState !== null) {
    if (nextDeps !== null) {
      const prevDeps: Array<mixed> | null = prevState[1];
      // 判断update前后value是否变化
      if (areHookInputsEqual(nextDeps, prevDeps)) {
        // 未变化
        return prevState[0];
      }
    }
  }

  // 变化，将新的callback作为value
  hook.memoizedState = [callback, nextDeps];
  return callback;
}

import { DependencyList } from "react";

// https://github.com/facebook/react/blob/c2034716a5bff586ab68c41a14139a535cbd788e/packages/react-re
export default function areHookInputsEqual(
  nextDeps: DependencyList,
  prevDeps: DependencyList
): boolean {
  if (nextDeps.length !== prevDeps.length) {
    return false;
  }
```

```
  for (let i = 0; i < prevDeps.length && i < nextDeps.length; i++) {
    if (Object.is(nextDeps[i], prevDeps[i])) {
      continue;
    }
    return false;
  }
  return true;
}
```

call的实现

```
Function.prototype.call = function(t) {
    var context = t || window; // 处理传null的情况
        context.fn = this;
    var res = context.fn(...Array.from(arguments).slice(1)) // es6
    delete context.fn;
    return res;
}
```


map与object的区别

Map	Object	
意外的键	`Map` 默认情况不包含任何键。只包含显式插入的键。	一个 `Object` 有一个原型，原型链上的键名有可能和你自己在对象上的设置的键名产生冲突。**备注：**虽然从 ES5 开始可以用 `Object.create(null)` 来创建一个没有原型的对象，但是这种用法不太常见。
键的类型	一个 `Map` 的键可以是**任意值**，包括函数、对象或任意基本类型。	一个 `Object` 的键必须是一个 `String` 或是 `Symbol` 。
键的顺序	`Map` 中的键是有序的。因此，当迭代的时候，一个 `Map` 对象以插入的顺序返回键值。	虽然 `Object` 的键目前是有序的，但并不总是这样，而且这个顺序是复杂的。因此，最好不要依赖属性的顺序。自 ECMAScript 2015 规范以来，对象的属性被定义为是有序的；ECMAScript 2020 则额外定义了继承属性的顺序。参见 OrdinaryOwnPropertyKeys 和 EnumerateObjectProperties 抽象规范说明。但是，请注意没有可以迭代对象所有属性的机制，每一种机制只包含了属性的不同子集。（`for-in` 仅包含了以字符串为键的属性；`Object.keys` 仅包含了对象自身的、可枚举的、以字符串为键的属性；`Object.getOwnPropertyNames` 包含了所有以字符串为键的属性，即
Map	Object	

		使是不可枚举的； `Object.getOwnPropertySymbols` 与前者类似，但其包含的是以 `Symbol` 为键的属性，等等。）
Size	`Map` 的键值对个数可以轻易地通过 `size` 属性获取。	`Object` 的键值对个数只能手动计算.
迭代	`Map` 是 可迭代的 的，所以可以直接被迭代。	`Object` 没有实现 迭代协议，所以使用 JavaSctipt 的 for...of 表达式并不能直接迭代对象。**备注：**对象可以实现迭代协议，或者你可以使用 `Object.keys` 或 `Object.entries` 。for...in 表达式允许你迭代一个对象的*可枚举*属性。
性能	在频繁增删键值对的场景下表现更好。	在频繁添加和删除键值对的场景下未作出优化。
序列化和解析	没有元素的序列化和解析的支持。（但是你可以使用携带 *replacer* 参数的 `JSON.stringify()` 创建一个自己的对 `Map` 的序列化和解析支持。参见 Stack Overflow 上的提问：How do you JSON.stringify an ES6 Map?）	原生的由 `Object` 到 JSON 的序列化支持，使用 `JSON.stringify()` 。原生的由 JSON 到 `Object` 的解析支持，使用

- 与 `Set` 相比， `WeakSet` 只能是**对象的集合**，而不能是任何类型的任意值。
- `WeakSet` 持弱引用：集合中对象的引用为弱引用。 如果没有其他的对 `WeakSet` 中对象的引用，那么这些对象会被当成垃圾回收掉。 这也意味着 WeakSet 中没有存储当前对象的列表。 正因为这样， `WeakSet` 是不可枚举的。

箭头函数没有原型， 所以没有call原型方法

jsonp的缺点

JSONP

- 通过添加一个

- 缺点：
  - 只支持GET请求 **且** 不安全 ，**可能遇到XSS攻击，不过它的好处是可以向老浏览器或不支持CORS的网站请求数据**
  - jsonp在调用失败的时候不会返回各种HTTP状态码
  - 支持http请求调用， 不支持两个不同域的页面之间通信
  - 安全性。万一假如提供jsonp的服务存在页面注入漏洞，即它返回的javascript的内容被人控制的。那么结果是什么？所有调用这个 jsonp的网站都会存在漏洞。于是无法把危险控制在一个域名下...所以在使用jsonp的时候必须要保证使用的jsonp服务必须是安全可信的。
- 好处：
  - 兼容性更好
  - 在请求完毕后可以通过调用callback的方式回传结果。将回调方法的权限给了调用方。这个就相当于将controller层和view层终于分开了。我提供的jsonp服务只提供纯服务的数据，至于提供服务以 后的页面渲染和后续view操作都由调用者来自己定义就好了。如果有两个页面需要渲染同一份数据，你们只需要有不同的渲染逻辑就可以了，逻辑都可以使用同 一个jsonp服务。

- ```js 复制代码
  const jsonp = ({ url, params, callbackName }) => {
    const generateURL = () => {
      let dataStr = '';
      for(let key in params) {
        dataStr += `${key}=${params[key]}&`;
      }
      dataStr += `callback=${callbackName}`;
      return `${url}?${dataStr}`;
    };
    return new Promise((resolve, reject) => {
      // 初始化回调函数名称
      callbackName = callbackName || Math.random().toString.replace(',', '');
      // 创建 script 元素并加入到当前文档中
      let scriptEle = document.createElement('script');
      scriptEle.src = generateURL();
      document.body.appendChild(scriptEle);
      // 绑定到 window 上，为了后面调用
      window[callbackName] = (data) => {
        resolve(data);
        // script 执行完了，成为无用元素，需要清除
  ```
  ```

```
        document.body.removeChild(scriptEle);
      }
    });
}

jsonp({
  url: 'http://localhost:3000',
  params: {
    a: 1,
    b: 2
  }
}).then(data => {
  // 拿到数据进行处理
  console.log(data); // 数据包
})

// 服务端
let express = require('express')
let app = express()
app.get('/', function(req, res) {
  let { a, b, callback } = req.query
  console.log(a); // 1
  console.log(b); // 2
  // 注意哦，返回给script标签，浏览器直接把这部分字符串执行
  res.end(`${callback}('数据包')`);
})
app.listen(3000)
```

js事件机制

捕获、冒泡、事件委托（事件委托利用了事件冒泡，只指定一个事件处理程序，就可以处理某一类型的所有事件。）

zhuanlan.zhihu.com/p/73091706

```
// 事件监听 EventTarget.addEventListener()
// addEventListener()基本上有三个参数，分别是「事件名称」、「事件的处理程序」(事件触发时执行的function)，以
```

GET POST

- GET 在浏览器回退时是无害的，而 POST 会再次发起请求
- **GET 请求会被浏览器主动缓存，而 POST 不会，除非手动设置**

- `GET` 请求参数会被安逗保留在浏览器历史记录里，而 `POST` 中的参数不会被保留
- **`GET` 请求在 `URL` 中传递的参数有长度限制(浏览器限制大小不同)，而 `POST` 没有限制**
- **`GET` 参数通过 `URL` 传递， `POST` 放在 `Request body` 中**
- `GET` 产生的URL地址可以被收藏，而 `POST` 不可以
- **`GET` 没有 `POST` 安全，因为 `GET` 请求参数直接暴露在 `URL` 上，所以不能用来传递敏感信息**
- **`GET` 请求只能进行 `URL` 编码，而 `POST` 支持多种编码方式**
- 对参数的数据类型， `GET` 只接受 `ASCII` 字符，而 `POST` 没有限制
- `GET` 产生一个TCP数据包， `POST` 产生两个数据包(Firefox只发一次)。GET浏览器把 http header和data一起发出去，响应成功200，POST先发送header，响应100 continue，再发送data，响应成功200

Cookie的相关属性

developer.mozilla.org/zh-CN/docs/...

expires/ max-age/httpOnly/Secure/Domain/Path/SameSite

- `SameSite` Cookie 允许服务器要求某个 cookie 在跨站请求时不会被发送，（其中 Site (en-US) 由可注册域定义），从而可以阻止跨站请求伪造攻击（CSRF）。
- `Domain` 和 `Path` 标识定义了 Cookie 的*作用域：*即允许 Cookie 应该发送给哪些 URL。
- expires/ max-age： cookie过期时间
- httpOnly/Secure：限制访问 Cookie

代理 proxy

用nodejs搭建代理服务器

- express中间件**http-proxy-middleware**
    - 1、target，指的是目标网站，或者被代理的网站。

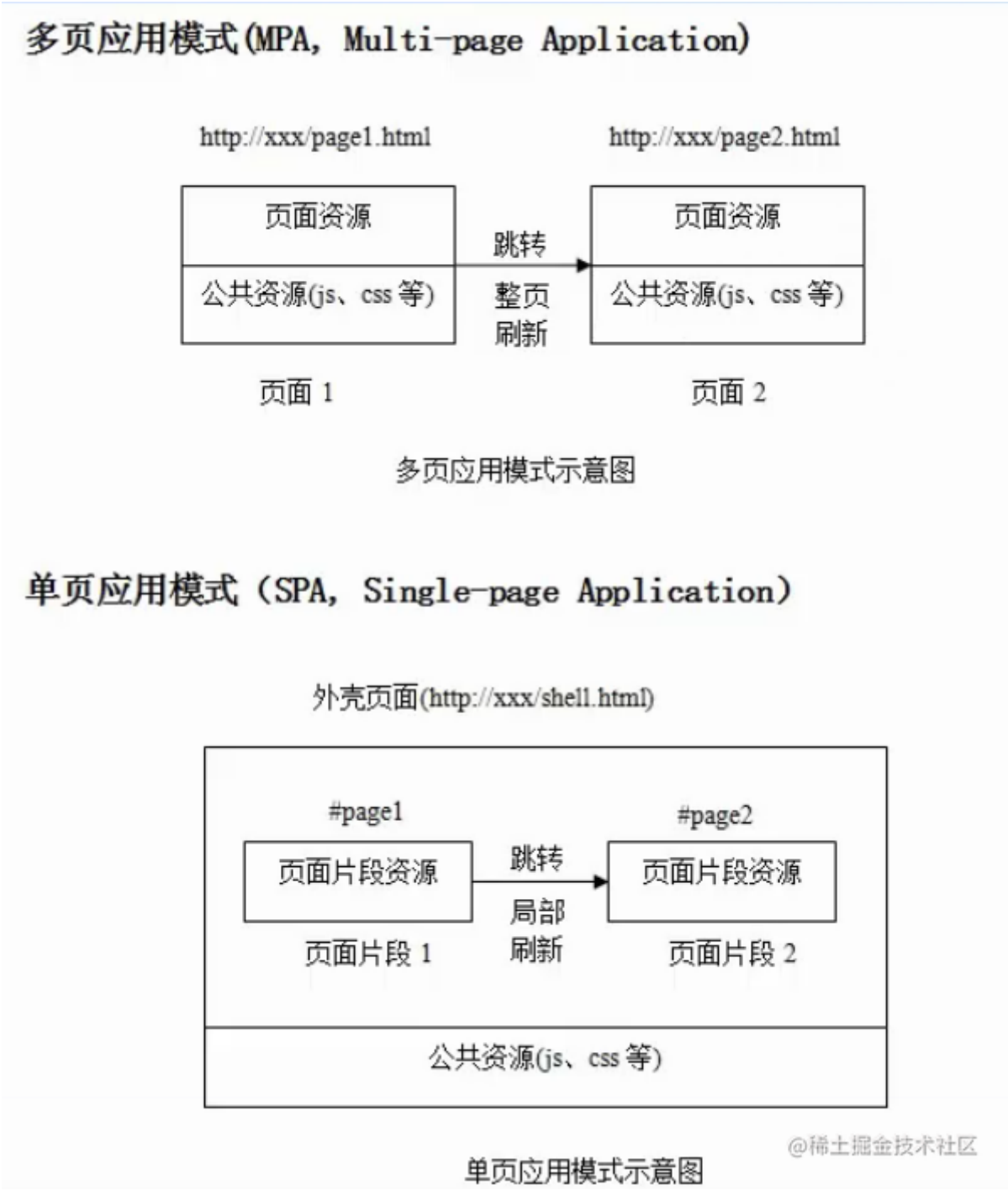
      2、changeOrigin是否更改host。默认为false，不重写。

      3、pathRewrite路径重写，这个特性看需求。

单页面和多页面的区别

单页面访问会有白屏等待事件： 加载bundle文件

多页面直接访问源文件,



SPA单页面应用（SinglePage Web Application），指只有一个主页面的应用（一个html页面），一开始只需要加载一次js、css的相关资源。所有内容都包含在主页面，对每一个功能模块组件化。单页应用跳转，就是切换相关组件，仅仅刷新局部资源。

MPA多页面应用（MultiPage Application），指有多个独立页面的应用（多个html页面），每个页面必须重复加载js、css等相关资源。多页应用跳转，需要整页资源刷新。

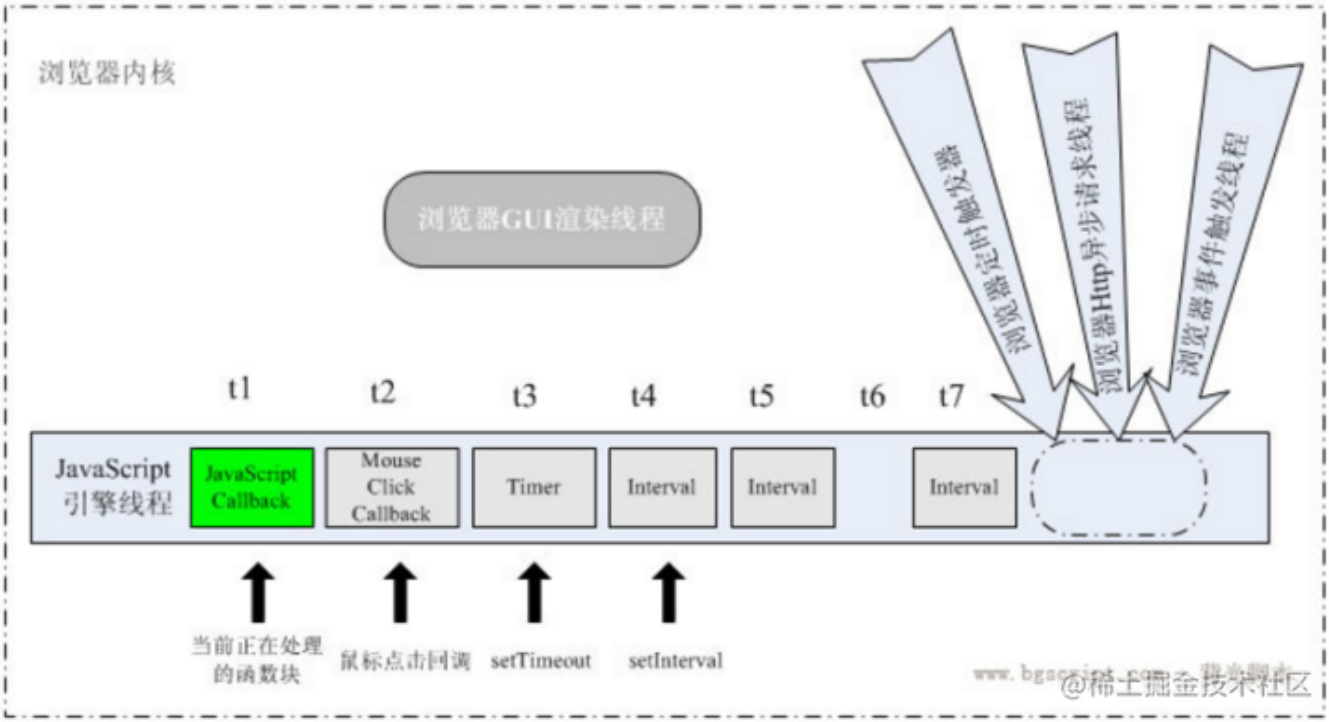
	单页面应用（SinglePage Web Application，SPA）	多页面应用（MultiPage Application，MPA）
组成	一个外壳页面和多个页面片段组成	多个完整页面构成
资源共用(css,js)	共用，只需在外壳部分加载	不共用，每个页面都需要加载
刷新方式	页面局部刷新或更改	整页刷新
url 模式	a.com/#/pageone a.com/#/pagetwo	a.com/pageone.html a.com/pagetwo.html
用户体验	页面片段间的切换快，用户体验良好	页面切换加载缓慢，流畅度不够，用户体验比较差
转场动画	容易实现	无法实现
数据传递	容易	依赖 url传参、或者cookie 、localStorage等
搜索引擎优化(SEO)	需要单独方案、实现较为困难、不利于SEO检索 可利用服务器端渲染(SSR)优化	实现方法简易
试用范围	高要求的体验度、追求界面流畅的应用	适用于追求高度支持搜索引擎的应用
开发成本	较高，常需借助专业的框架	较低，但页面重复代码多
维护成本	相对容易	相对复杂

js遇到异步函数， 挂起执行的话是交给浏览器其他线程处理

zhuanlan.zhihu.com/p/23659122： *参考*

- *定时器函数setTimeout（交给定时器线程处理）*
- *事件绑定（onclick、onkeydown....,交给事件触发线程处理）*
- *AJAX（交给HTTP线程处理）*
- *回调函数*

js是单线程语言，浏览器只分配给js一个主线程，用来执行任务（函数），但一次只能执行一个任务，这些任务形成一个任务队列排队等候执行，但前端的某些任务是非常耗时的，比如网络请求，定时器和事件监听，如果让他们和别的任务一样，都老老实实的排队等待执行的话，执行效率会非常的低，甚至导致页面的假死。所以，**浏览器为这些耗时任务开辟了另外的线程，主要包括http请求线程，浏览器定时触发器，浏览器事件触发线程，这些任务是异步的。**下图说明了浏览器的主要线程。



再说说任务队列

刚才说到浏览器为网络请求这样的异步任务单独开了一个线程，那么问题来了，这些异步任务完成后，主线程怎么知道呢？答案就是回调函数，整个程序是事件驱动的，每个事件都会绑定相应的回调函数，举个栗子，有段代码设置了一个定时器

```js
setTimeout(function(){
    console.log(time is out);
}，50）；
```

执行这段代码的时候，浏览器异步执行计时操作，当50ms到了后，会触发定时事件，这个时候，就会把回调函数放到任务队列里。整个程序就是通过这样的一个个事件驱动起来的。 所以说，js是一直是单线程的，浏览器才是实现异步的那个家伙。

错误监控

promise 错误

```js
window.addEventListener('unhandledrejection', function(event) {
  // 这个事件对象有两个特殊的属性：
  alert(event.promise); // [object Promise] - 生成该全局 error 的 promise
  alert(event.reason); // Error: Whoops! - 未处理的 error 对象
});
```

react 顶层 **componentDidCatch ** **getDerivedStateFromError**

```react
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // 更新 state 使下一次渲染能够显示降级后的 UI
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
    // 你同样可以将错误日志上报给服务器
    logErrorToMyService(error, errorInfo);
  }

  render() {
    if (this.state.hasError) {
      // 你可以自定义降级后的 UI 并渲染
      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
  }
}
```

Redux 原则和理念

redux 的设计理念

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:

- **1、唯一数据源**

  整个应用的数据存储在一个统一的状态树中，也就是我们前面所说的公共的store 文件。在组件都会从这个store中获取数据。

  2、保持只读状态

  state是只读的，唯一改变state的方法就是触发action，action是一个用于描述以发生时间的普通对象。

  3、数据改变只能通过纯函数来执行

  使用纯函数来执行修改，为了描述action如何改变state的，你需要编写reducers。

Typescript 的高级使用

把 某一种类型 去掉： keyof , Pick , key的类型判断

tsx 复制代码

```tsx
type FilterKeys<T, U> = {
  [key in keyof T]: T[key] extends U ? never : key;
}[keyof T]

type Fix<T, U> = Pick<T, FilterKeys<T, U>>;

type A1 = Fix<IType, string>;
```

手写代码

1. 数组扁平化

2. 数值 3个位一组切割 12,233,424

所有任务同时执行；有依赖需要相继执行；给出最后执行时间

```
import "./styles.css";

(document.getElementById("app") as HTMLElement).innerHTML = `
<div>
所有任务同时执行；有依赖需要相继执行；给出最后执行时间
</div>
`;
interface ITask {
  name: string;
  time: number;
  dependency: string;
}
//例如下面这些任务执行总时间为 7
const tasks: ITask[] = [
  {
    name: "task2",
    time: 2,
    dependency: ""
  },
  {
    name: "task3",
    time: 3,
    dependency: "task1"
  },
  {
    name: "task4",
    time: 3,
    dependency: "task3"
  },
  {
    name: "task5",
    time: 4,
    dependency: "task2"
  },
  {
    name: "task1",
    time: 1,
    dependency: ""
  }
]
function getTime(tasks: ITask[]) {
  const notDependency = tasks.filter((d)=> !d.dependency)
  const map = {};
  //
  notDependency.forEach(function(d){
    const name = d.name;
    const findTasks = tasks.filter(t => t.dependency === name)
    map[d.name] = d.time; // 类似执行
    findTasks.map(function(find){
```

```
      map[find.name] = (map[d.name] || d.time) + find?.time
    });
})
  const surplus = tasks.filter(d => !map[d.name])
  surplus.map(d => map[d.name] = map[d.dependency] + d.time)
  return map;
}

console.log(getTime(tasks));
```