# 基础

## html

### **doctype的作用是什么**

DOCTYPE是html5**标准网页声明**，且必须声明在HTML文档的第一行。**来告知浏览器的解析器用什么文档标准解析这个文档**。

如果删除调这个标签，则代表以怪异模式 渲染页面，它所识别的语法向前靠拢，尽量兼容以前的版本,否则就是 标准模式

### **你对HTML语义化的理解**

语义化是指使用恰当语义的html标签，让页面具有良好的结构与含义，比如<p>标签就代表段落，<article>代表正文内容,<header> <nav> <aside> <footer>等。

优点:

* 增强开发维护可读性
* 更适合机器解析SEO，生成目录,搜索引擎爬虫等

### **HTML5的新特性**

更加语义化的元素。 article、footer、header、nav、section

本地化储存。 localStorage 和 sessionStorage

拖曳以及释放的api。 Drag and drop

媒体播放。 video 和 audio

多任务。 webworker

全双工通信协议。 websocket

历史管理 history

跨域资源共享（CORS） Access-Control-Allow-Origin

页面可见性改变事件 visibilitychange

跨窗口通信 PostMessage

绘画：canvas

### **src和href的区别**

* src所指定的资源会被下载并嵌入到文档，浏览器会等待src的解析执行
* href一般用于超链接，如果这个超链接内有资源的下载，浏览器不会因他停止渲染

### **有哪些常用的meta标签**

meta标签用来描述一个HTML网页文档的属性，告诉机器该如何解析这个页面，常见属性为name、content、http-equiv、charset

* charset，用于描述HTML文档的编码形式
* name，用于浏览器解析，可以对浏览器的一些兼容性问题做处理（eg：告诉浏览器用webkit内核进行解析
* http-equiv，用于添加http头部内容

### **script标签中async和defer的区别**

首先浏览器自上而下的解析html的标签，且是同步的方式。即遇到script标签事，会阻塞html解析器，下载script标签内容并执行，由于浏览器是自上而下的解析html ，所以script标签以上的dom是可用的，以下的dom是不可用的。

而async和defer是异步加载script标签(不过对于内敛script无用)，不会阻塞html解析器，不同点是：

1. async在多个script脚本存在时，不会按照顺序执行，谁先下载好就执行谁
2. defer需要按顺序执行，因为会等到documentcontentloaded以后，再去执行

### **html和xhtml的区别**

XHTML 元素是以 XML 格式编写的 HTML 元素。XHTML是严格版本的HTML，例如它要求标签必须小写，标签必须被正确关闭，标签顺序必须正确排列，对于属性都必须使用双引号等

* XHTML 得到所有主流浏览器的支持
* XHTML 与 HTML 4.01 几乎是相同的
* XHTML 是更严格更纯净的 HTML 版本

## js原生系列

### **1、执行上下文**

指当前执行环境中的变量、函数声明，参数（arguments），作用域链，this等信息。分为全局执行上下文、函数执行上下文，其区别在于全局执行上下文只有一个，函数执行上下文在每次调用函数时候会创建一个新的函数执行上下文。

#### 1.1 js运行过程

* 语法分析
* 预编译
* 解释执行

语法分析很简单，就是引擎检查你的代码有没有什么低级的语法错误； 解释执行顾名思义便是执行代码了； 预编译简单理解就是在内存中开辟一些空间，存放一些变量与函数 ；

#### 1.2 预编译之前需要了解的

* 函数声明整体提升(函数声明永远提升到js文件最前面)
* 变量 声明提升
* 任何变量,如果变量未经声明就赋值,归全局所有

#### 1.3 预编译过程

**函数预编译**

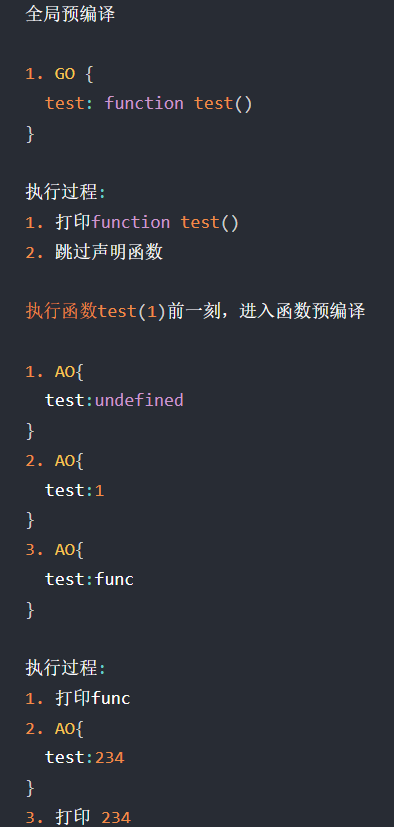
预编译发生在函数执行的前一刻

* 创建 AO 对象 --- 执行期上下文对象(Activation Object)
* 找形参和变量声明，将变量和形参名作为 AO 属性名,值为 undefined (变量提升)
* 将实参值和形参相统一
* 在函数里面找函数声明,将函数名作为属性值挂上，值为函数体
* 在这过程中还有两个步骤：arguments 作为属性，值为实参数组，this作为属性，this为 window

**全局预编译**

全局预编译过程相比函数预编译过程少了第三步，没有参数的概念。如果又存在全局又存在函数则首先考虑全局在考虑函数预编译。执行代码过程中如果AO有该变量，就先取AO的，没有再去GO中取。

* 创建 GO 全局对象(GO === window)，Global Object
* 寻找声明变量，赋值undefined
* 寻找函数声明，函数名作为属性名，函数体作为值
* this作为属性，值为window



### **2、作用域和作用域链**

作用域存储了执行期上下文的集合

[[scope]]: js的函数的可以视为一个对象，对象中有一些属性我们可以访问，但有些不可以，这些属性仅供js引擎存取，[[scope]]就是其中之一，[[scope]]指的就是我们所说的作用域，其中存储了执行期上下文的集合

作用域链: [[scope]]中所存储的执行期上下文对象的集合层链式链接，我们把这种链式链接叫做作用域链

### **3、闭包**

一句话可以概括:闭包就是能够读取其他函数内部变量的函数，或者子函数在外调用，子函数所在的父函数的作用域不会被释放。

从实践角度上来说，只有满足

* 即使创建它的上下文已经销毁，它仍然存在，
* 在代码中引入了自由变量，才称为闭包

闭包的应用：

* 模仿块级作用域。
* 保存外部函数的变量。
* 封装私有变量

闭包优点：

* 1:变量长期驻扎在内存中；
* 2:避免全局变量的污染；
* 3:私有成员的存在 ；

闭包缺点：

1. 由于闭包会携带包含它的函数的作用域，所以闭包所占用的内存会比普通函数更大，过度使用闭包可能导致内存占用过多

对于ie9之前的版本，如果作用域中保存着一个html元素，那么就意味着这个元素无法被销毁。

以下方式可以回收

2）如果使用不当可能导致内存泄漏（使用闭包的同时比较容易形成循环引用，如果闭包的作用域链中保存着一些DOM节点，这时候就有可能造成内存泄漏。但是这本身并非闭包的问题，也并非javascript的问题。在IE浏览器中由于BOM与DOM中的对象是使用C++以COM对象的方式实现的，而COM对象的垃圾回收机制是采用引用计数策略。在基于引用计数策略的垃圾回收机制中，如果两个对象之间形成了循环引用，那么这两个对象都无法被回收，但循环引用造成的内存泄漏在本质上也不是闭包造成的。

同样，如果要解决循环引用带来的内存泄漏问题，我们只需要在循环引用中的变量设为null即可。将变量设置为null意味着切断变量与它此前引用的值之间的连接。当垃圾收集器下次运行时，就会删除这些值并回收它们占用的内存。）

闭包模仿块级作用域

闭包实现私有变量

### **4、立即执行函数**

此类函数只会执行一次，并且直接销毁内存，不会占用内存空间，常常用来初始化数据

规则：

* 只有表达式才会执行
* 被执行符号执行的表达式会自动放弃表达式的名字，所以这种情况下，就相当于立即执行函数：



### **5、堆和栈**

栈：由操作系统自动分配释放 ，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈，比如js中函数执行栈。

堆： 一般由程序员分配释放， 若程序员不释放，程序结束时可能由操作系统回收，分配方式类似于链表。

根据缓存方式

栈：使用的是一级缓存，他们通常都是被调用时处于储存空间中，调用完毕立即释放。比如js中调用一个函数之后，立即释放执行栈，优点是存取速度快，但缺点是，存在栈中的数据大小与生存期必须是确定的，缺乏灵活性；

堆：则是存放在二级缓存中，生命周期由虚拟机的垃圾回收算法来决定，需要人为干预释放，所以调用这些对象的速度要相对来得低一些，由于要在运行时动态分配内存，存取速度较慢。

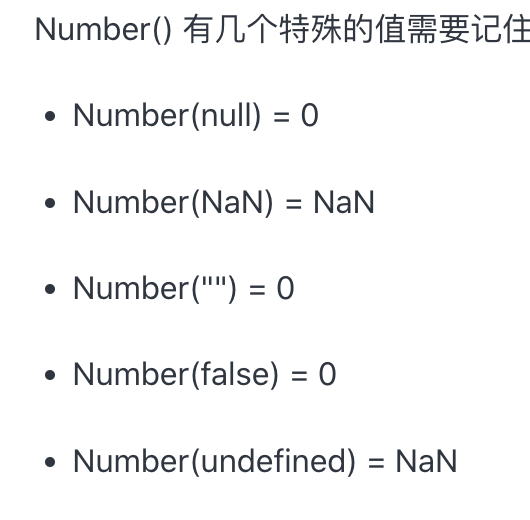
### **6、js类型判断**

JS类型判断有四种方法

* typeof操作符:判断值类型、函数类型、引用类型（具体引用类型无法判断）
* instanceof：instanceOf 的原理很简单，例如 a instanceOf b 就是利用 a 的隐式原型\_\_proto\_\_顺着原型链向上查找，如果过程中查到隐式原型的值等于 b 的显示原型 prototype 就返回 true,否则返回 false，如果一直找不到等于 null，也返回 false
* constructor：主要用于引用类型
* Object.prototype.toString .call()：toString() 是 Object 的原型方法，调用该方法，默认返回当前对象的 [[Class]] 。这是一个内部属性，其格式为 [object Xxx] ，其中 Xxx 就是对象的类型。

### **7、显式类型转换**

* Number(mix)



* parseInt(string,raidx)
* parseFloat(string)
* toString()，undefined和null不能使用这个方法，因为没有原型，如果添加参数则代表转为目标进制
* Boolean js种六种值转为false (undefined ,null ,'' ,NAN ,0,false),其他的都是true

### **隐式类型转换**

<https://chinese.freecodecamp.org/news/javascript-implicit-type-conversion/>

对于运算符：减乘除两侧的变量会转成number类型；对于+，如果是一侧存在string类型就会转为string类型、如果一侧number另一侧引用就会转字符串再拼接、其余情况转number

对于==：

* nan的比较永远为false
* 一侧存在boolean会转number
* 一侧存在string一侧存在number会转number
* 一侧原始类型一侧引用类型，会转原始类型（通过tostring）

### **new**

* new做的事情：
* 创建一个空对象
* 该对象的原型指向构造函数的原型
* 构造函数的this指向新创建的空对象，并执行函数返回结果
* 判断返回的结果是否是对象，如果是就返回结果，否则就返回新创建的对象

实现Object.creat()

### **继承**

#### 10.1 原型链继承

将父类实例作为子类的原型（ps：所有函数的默认原型都是Object实例

缺点：

* 属性被共用，无法实现多继承
* 子类实例化时无法向父类的构造函数传参

#### 10.2 构造函数继承

使用父类构造函数来增强子类

缺点：

* 每次都要新创建子类都会实例化一个父类，耗性能
* 不能继承父类原型上的属性和方法

#### 10.3 组合继承

缺点：调用了两次父类的构造函数，比较耗内存

#### 10.4 原型式继承

有一个对象，想在它的基础上再创建一个新对象。你需要把这个对象先传给object() ，然后再对返回的对象进行适当修改。类似于 Object.create()只传第一个参数的时候，本质上就是对传入的对象进行了一次浅复制,缺点就是新实例的属性都是后面添加的,无法复用

#### 10.5 寄生式组合继承

使用寄生式继承来继承父类原型，然后将返回的新对象赋值给子类原型。

* 第一步是创建超类型原型的一个副本
* 第二步是为创建的副本添加 constructor 属性，从而弥补因重写原型而失去的默认的 constructor 属性
* 最后一步，将新创建的对象（即副本）赋值给子类型的原型

### **this指向问题、call、bind、apply**

#### 11.1 this指向问题

* 预编译过程中,this指向全局对象window;
* 在严格模式下"use strict",为undefined.
* 对象的方法里调用,this指向调用该方法的对象.谁调用它就指向谁
* 构造函数里的this,指向创建出来的实例.

#### 11.2 call/apply/bind实现

apply

call

bind

#### 11.3 深拷贝

### **dom**

#### 可视区窗口尺寸

#### 滚动条

### **事件循环机制**

#### 13.1 浏览器时间循环机制

JS脚本scrpit包含异步任务和同步任务，首先同步任务在主线程上执行，当同步任务执行完成后，JS将当前执行栈清空。然后开始执行异步任务，异步任务分为宏任务和微任务，异步任务会放入任务队列当中，宏任务放入宏任务队列，微任务会放入微任务队列。

同步任务执行完之后，先去微任务队列拿出一个微任务，如果执行过程产生了微任务或者宏任务，会将他们放入各自的任务队列。循环往复，直到把微任务队列清空。

紧接着会从宏任务队列中拿出一个宏任务，执行过程中若产生了微任务，首先会把微任务放入微任务队列，然后等这个宏任务执行完成之后就会立即去执行这个微任务，如果没有产生微任务就继续执行下一个宏任务，循环往复，直到清空宏任务。

在宏任务执行完成前的一刻，渲染进程会触发GUI和dom的渲染。

* 微任务包括：process.nextTick ，promise ，MutationObserver。
* 宏任务包括：script ， setTimeout ，setInterval ，setImmediate ，I/O ，UI rendering。

#### 13.1 node循环机制

同步 =》 nextTick =》 异步微任务 =》 timer =》 io（setImmdiate =》timer）

### **Promise**

基于观察者模式的

#### 14.1 Promise.all

#### 14.2 Promise.race

#### 14.3 Promise.done

promise对象的回调链，不管以then还是reject结尾，最后一个方法如果报错就无法被捕捉到，因为promise的错误不会冒泡到全局，所以需要一个方法总是在回调链的尾端，保证无论抛出任何可能的错误。

#### 14.4 Promise.finally

#### 14.5 Promise.retry

### **垃圾回收**

**标记清除**

运行的时候给所有存储在内存中的变量都加上标记，然后，环境中的变量以及被环境中的变量引用的变量会被去掉标记，在此之后再被加上标记的变量将被视为准备删除的

可以使用任何方式标记标量：

* 通过翻转某个特殊位来记录一个变量何时进入环境
* 使用一个“进入环境”的标量列表和一个“离开环境”的变量列表来跟踪哪个变量发生了变化

**引用计数**

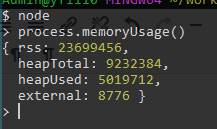
跟踪记录每个值被引用的次数，当引用值的次数归零，就认为没办法再访问这个变量，就会将其占用的空间回收

存在的问题：循环引用，对象a中包含一个指向对象b的指针，而对象b中也包含指向对象a的指针

避免方式：用完置为null，又叫接触引用

### **如何监控内存泄漏**

1. Chrome提供的开发者工具中有内存这一项，可以查看profile，也可以进行快照分析（以及开发者工具中的performance进行快照分析）
2. node的情况，提供了process.memoryUsage()可以查看内存占用情况



### **e.target和e.currentTarget的区别**

<https://www.jianshu.com/p/1dd668ccc97a?u_atoken=8834c3b2-973a-42ec-a27c-b5d0be54943a&u_asession=01-55ZsQdvf-qClntqYtc6tqR-by5H51GsyFSmGSEv51mae8wxpcPhEGexBcJzHUVDX0KNBwm7Lovlpxjd_P_q4JsKWYrT3W_NKPr8w6oU7K-7ulkbx3cUzoiUKCB7L1qIPMqDvQo0pEVbhSjSW3HVJmBkFo3NEHBv0PZUm6pbxQU&u_asig=05xDfG2WcTMjGHcg-i3G_3xZt6Wwbtbpj1mO9vTmMSRJwNYtawzZzmzxjXaPf8D4xp5wGOmPtj0NOHWukN5YtRFX4uZssSi0hUh9epvbWT9xAA5SK-vTzxWx1cewlp6gE9lGZyvjLmXLZKBjHRiKMQjRzx4k5DHOlCjJqjff3ExO_9JS7q8ZD7Xtz2Ly-b0kmuyAKRFSVJkkdwVUnyHAIJzVrivau3swy41y6X1akwx2dRyFQmNp_hWZZ8vDuCHcY8OF33MttdlYrQH7V14NMYIe3h9VXwMyh6PgyDIVSG1W8sIsB99Gcw45KKIlavC_3RQKranM7rBKraCitp69DNzHD8eRKTClQ-a0hmdKdZ-jMKXVRsntWxT8kcMGF8pXi1mWspDxyAEEo4kbsryBKb9Q&u_aref=KOwXmJ2JMwHhPDOJJGkUSVDP9YI%3D>

target为事件具体触发的对象，currentTarget是注册事件的对象

### **事件委托**

**内存和性能**：在js中，添加到页面的事件处理程序会占用内存、直接影响到页面的性能，会导致页面交互就绪时间延迟

利用事件冒泡，指定一个事件处理程序就可以管理某一类型的所有事件

并且跟踪的事件处理程序越少，移除他们就越容易

### **柯里化**

柯里化（Currying）是把接受多个参数的函数变换成接受一个单一参数(最初函数的第一个参数)的函数，并且函数返回值接受余下的参数且返回结果的新函数的技术。

### **定时器**

关于定时器，指定的时间间隔标识何时将定时器的代码添加到队列中，而不是何时实际执行代码

重复定时器的规则有两个问题：1）某些间隔会被跳过 2）多个定时器的代码执行之间的间隔可能会比预期的小

解决办法：链式调用setTimeout

* 在前一个定时器代码执行完之前，不会向队列插入新的定时器代码，确保不会有任何丢失的间隔
* 可以保证下一次定时器代码执行之前，至少要等待指定的间隔，避免连续运行

### **防抖节流**

1、防抖：防抖是指在事件触发n秒后再执行回调，如果在n秒内再次被触发，则重新计算时间。

2、节流：节流是指如果持续触发某个事件，则每隔n秒执行一次。

### **事件流**

描述的是从页面接收事件的顺序

dom2把事件流分成了三个阶段：事件捕获阶段、处于目标阶段、事件冒泡阶段

阻止冒泡：

window.event.cancleBubble = true(IE谷歌阻止冒泡)

e.preventDefault()(火狐谷歌阻止冒泡)

### **0.1+0 .2为什么不等于0.3**

https://juejin.cn/post/6844903680362151950

精度损失可能出现在进制转化和对阶运算过程中

0.1和0.2被转成二进制后会无限循环，而js内的最大安全数不能超过52位，所以截取之后再做运算出现了精度损失

一个直接的解决方法就是设置一个误差范围，通常称为“机器精度”。对JavaScript来说，这个值通常为2-52，在ES6中，提供了Number.EPSILON属性，而它的值就是2-52，只要判断0.1+0.2-0.3是否小于Number.EPSILON，如果小于，就可以判断为0.1+0.2 ===0.3

解决办法：1个办法是使用整型代替浮点数计算；2是不要直接比较两个浮点数，而应该使用bignumber.js这样的浮点数运算库

### **bigint**

BigInt 是一种内置对象，它提供了一种方法来表示大于 2^53 - 1 的整数。这原本是 Javascript 中可以用 [Number](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Number) 表示的最大数字。BigInt 可以表示任意大的整数。

可以用在一个整数字面量后面加 n 的方式定义一个 BigInt ，如：10n，或者调用函数 BigInt()（但不包含 new 运算符）并传递一个整数值或字符串值。

### **实现reduce**

### **用reduce实现map**

### **长列表渲染问题**

<https://juejin.cn/post/6995334008603148295>h

## es6

### **箭头函数和普通函数区别**

* 语法更加简洁
* 箭头函数不会创建自己的this，this在定义时，会从自己作用域链的上一层继承this
* call、apply、bind无法改变箭头函数this指向
* 箭头函数不能用作构造函数使用

是因为箭头函数没有`prototype`与内置的[[Construct]]方法，无法完成构造。

* 箭头函数没有自己的arguments
* 箭头函数并不能用作generator函数、不能使用yeild关键字

### **对象的枚举**

描述对象的enumerable属性，称为“可枚举性”，如果该属性为false，就表示某些操作会忽略当前属性。

目前，有四个操作会忽略enumerable为false的属性。

* for...in循环：只遍历对象自身的和继承的可枚举的属性。
* Object.keys()：返回对象自身的所有可枚举的属性的键名。
* JSON.stringify()：只串行化对象自身的可枚举的属性。
* Object.assign()： 忽略enumerable为false的属性，只拷贝对象自身的可枚举的属性。

### 属性的遍历

ES6 一共有 5 种方法可以遍历对象的属性。

**（1）for...in**

for...in循环遍历对象（含自身属性、含继承属性、不含 Symbol 属性、不含不可枚举属性）。

**（2）Object.keys(obj)**

Object.keys返回一个数组遍历对象（含自身属性、不含继承属性、不含 Symbol 属性、不含不可枚举属性）。

**（3）Object.getOwnPropertyNames(obj)**

Object.getOwnPropertyNames返回一个数组，（含自身属性、不含继承属性、不含 Symbol 属性、含不可枚举属性）。

**（4）Object.getOwnPropertySymbols(obj)**

Object.getOwnPropertySymbols返回一个数组，包含对象自身的所有 Symbol 属性的键名。（含自身属性、不含继承属性、含 Symbol 属性、含不可枚举属性）。

**（5）Reflect.ownKeys(obj)**

Reflect.ownKeys返回一个数组，（含自身属性、不含继承属性、含 Symbol 属性、含不可枚举属性）。

以上的 5 种方法遍历对象的键名，都遵守同样的属性遍历的次序规则。

* 首先遍历所有数值键，按照数值升序排列。
* 其次遍历所有字符串键，按照加入时间升序排列。
* 最后遍历所有 Symbol 键，按照加入时间升序排列。

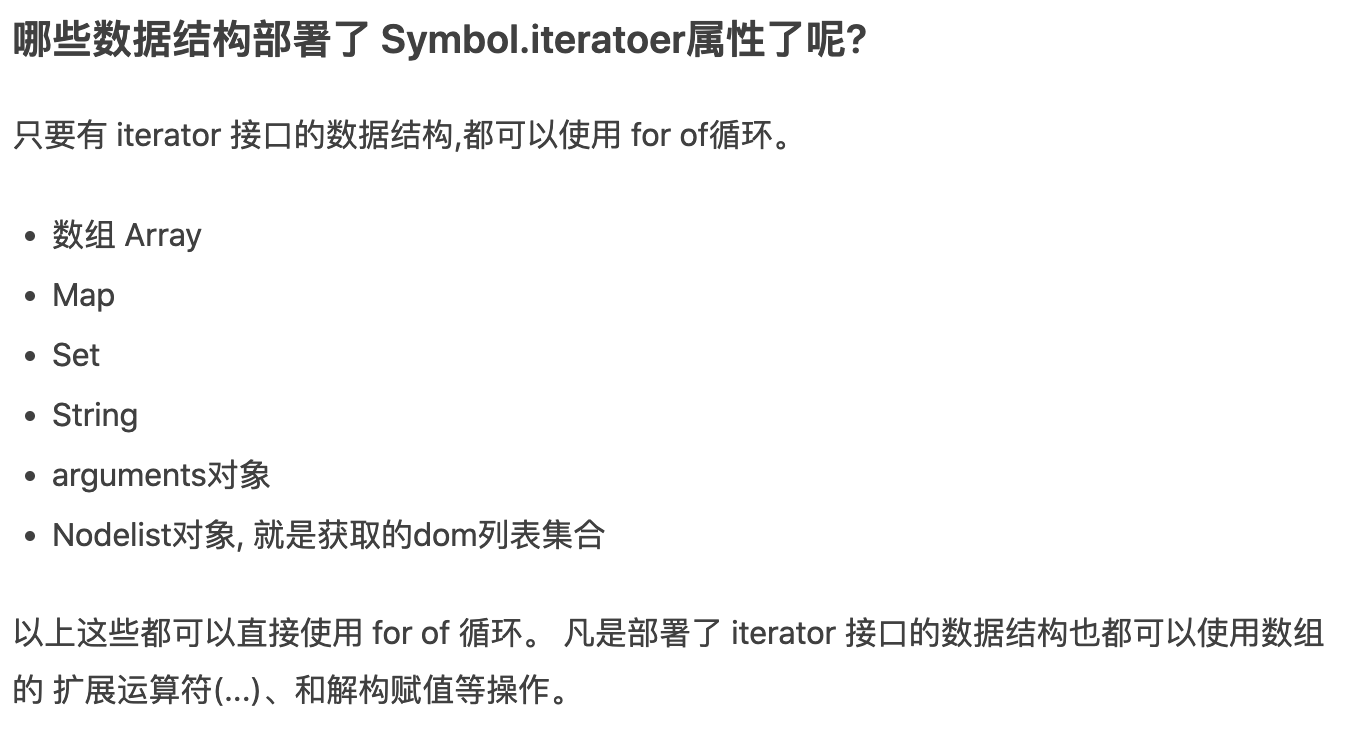
### **Symbol**

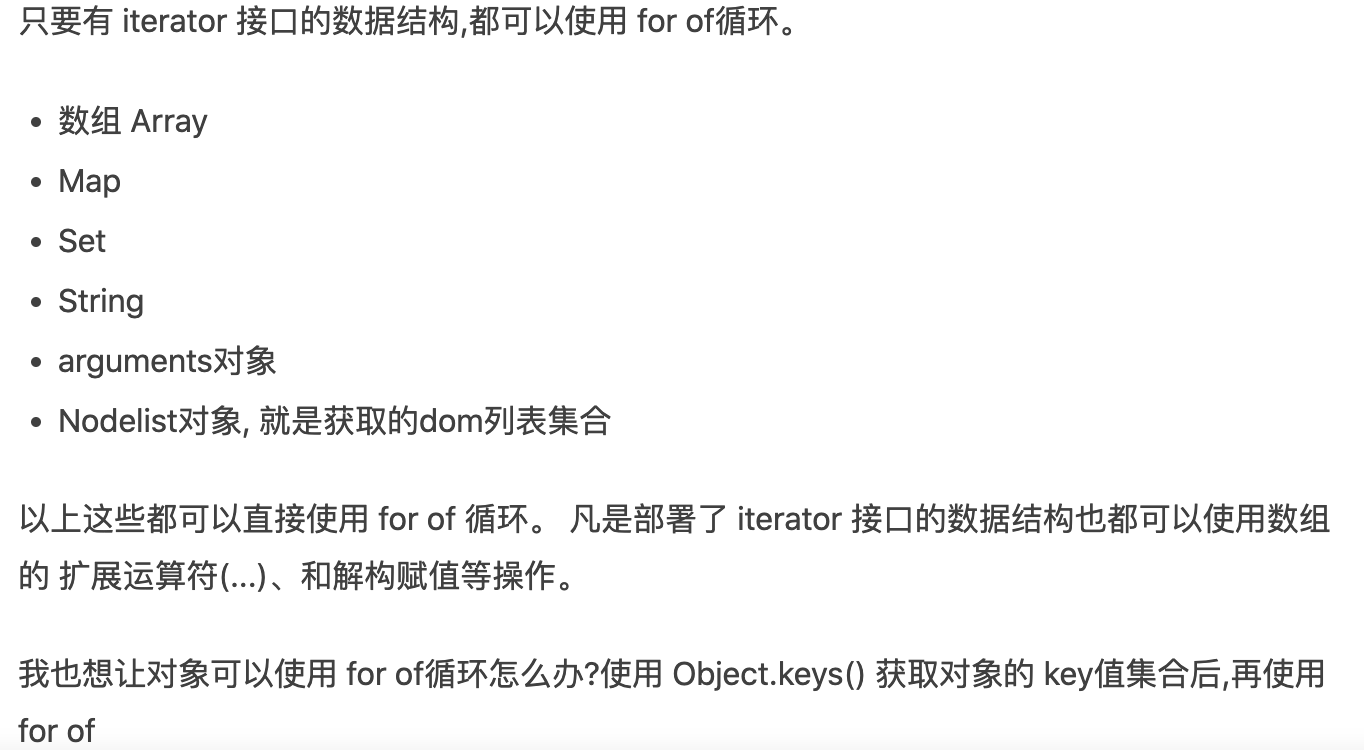
symbol是第七种原始数据类型，给对象添加新方法又怕新方法与现有方法名字冲突，就可以使用Symbol。

Symbol.for()接受一个字符串作为参数，然后搜索有没有以该参数作为名称的 Symbol 值。如果有，就返回这个 Symbol 值，否则就新建一个以该字符串为名称的 Symbol 值，并将其注册到全局。

Symbol的内置值







### **set&map**

**set**

set 属性方法：constructor、size、add、delete、has、clear

set遍历操作：keys、values、entries、forEach

weakset特点：

1）接受数组或类数组作为参数

2）方法包括：size、add、delete、has

WeakSet为弱引用，垃圾回收机制不考虑weakset的回收问题，当weakset中的对象不再被其他对象引用，则垃圾回收机制就会自动回收该对象所占用的内存，而不考虑是否被weakset所引用

weakset不能遍历原因：是因为成员都是弱引用，随时可能消失，遍历机制无法保证成员的存在，很可能刚刚遍历结束，成员就取不到了。

weakset用途：WeakSet的一个用处，是储存DOM节点，而不用担心这些节点从文档移除时，会引发内存泄漏

**map**

map属性方法：size、set、get、has、delete、clear

map遍历操作：keys、values、entries、forEach

weakmap特点：

1. 只接受对象作为键名
2. 不能遍历，没有size
3. 没有clear
4. 方法包括：set、get、has、delete

它的键名所引用的对象都是弱引用，即垃圾回收机制不将该引用考虑在内。因此，只要所引用的对象的其他引用都被清除，垃圾回收机制就会释放该对象所占用的内存。也就是说，一旦不再需要，WeakMap 里面的键名对象和所对应的键值对会自动消失，不用手动删除引用。

weakmap用途：dom节点作为键名，当dom被清除，map里键值对就会消失，不存在内存泄漏风险

### **proxy**

概述：Proxy 可以理解成，在目标对象之前架设一层“拦截”，外界对该对象的访问，都必须先通过这层拦截，因此提供了一种机制，可以对外界的访问进行过滤和改写。

提供的拦截类型：

* **get(target, propKey, receiver)**：拦截对象属性的读取，比如proxy.foo和proxy['foo']。
* **set(target, propKey, value, receiver)**：拦截对象属性的设置，比如proxy.foo = v或proxy['foo'] = v，返回一个布尔值。
* **has(target, propKey)**：拦截propKey in proxy的操作，返回一个布尔值。
* **deleteProperty(target, propKey)**：拦截delete proxy[propKey]的操作，返回一个布尔值。
* **ownKeys(target)**：拦截Object.getOwnPropertyNames(proxy)、Object.getOwnPropertySymbols(proxy)、Object.keys(proxy)、for...in循环，返回一个数组。该方法返回目标对象所有自身的属性的属性名，而Object.keys()的返回结果仅包括目标对象自身的可遍历属性。
* **getOwnPropertyDescriptor(target, propKey)**：拦截Object.getOwnPropertyDescriptor(proxy, propKey)，返回属性的描述对象。
* **defineProperty(target, propKey, propDesc)**：拦截Object.defineProperty(proxy, propKey, propDesc）、Object.defineProperties(proxy, propDescs)，返回一个布尔值。
* **preventExtensions(target)**：拦截Object.preventExtensions(proxy)，返回一个布尔值。
* **getPrototypeOf(target)**：拦截Object.getPrototypeOf(proxy)，返回一个对象。
* **isExtensible(target)**：拦截Object.isExtensible(proxy)，返回一个布尔值。
* **setPrototypeOf(target, proto)**：拦截Object.setPrototypeOf(proxy, proto)，返回一个布尔值。如果目标对象是函数，那么还有两种额外操作可以拦截。
* **apply(target, object, args)**：拦截 Proxy 实例作为函数调用的操作，比如proxy(...args)、proxy.call(object, ...args)、proxy.apply(...)。
* **construct(target, args)**：拦截 Proxy 实例作为构造函数调用的操作，比如new proxy(...args)。

this问题

 Proxy 代理的情况下，目标对象内部的this关键字会指向 Proxy 代理。

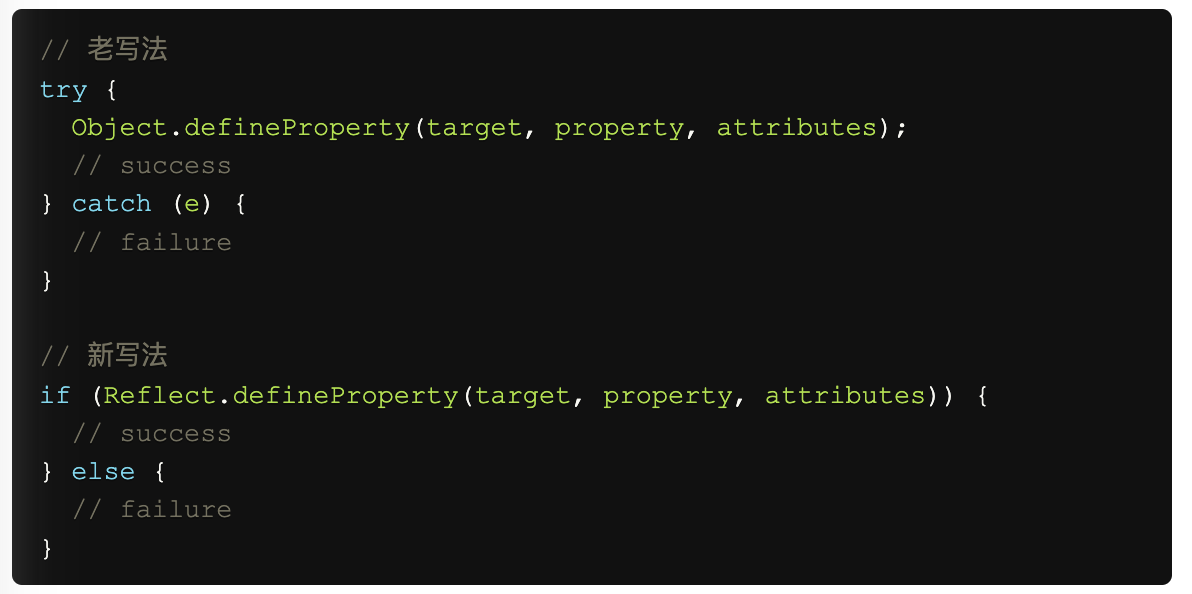


### **Reflect**

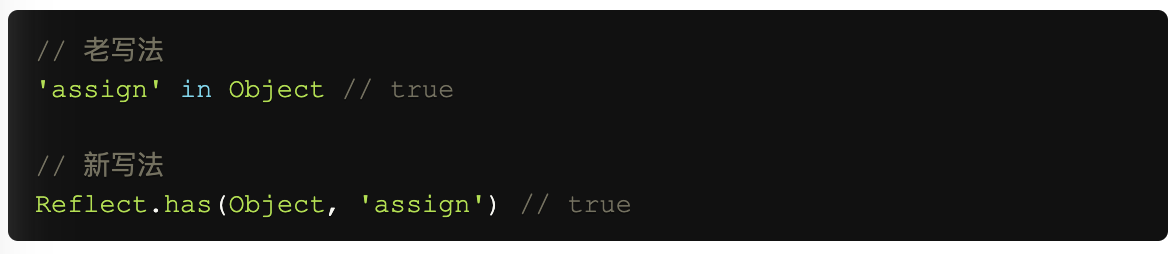
**设计目的：**

1）将Object对象的一些明显属于语言内部的方法放到Reflect对象上；

2）修改某些Object方法的返回结果，让其变得更合理。



3）让Object操作都变成函数行为。



1. Reflect对象的方法与Proxy对象的方法一一对应，只要是Proxy对象的方法，就能在Reflect对象上找到对应的方法。也就是说，不管Proxy怎么修改默认行为，你总可以在Reflect上获取默认行为。

**使用proxy实现观察者模式（实现以下效果）**

****

实现方式

### **Iterator和for of**

模拟interator实现

原生具备 Iterator 接口的数据结构如下。

* Array
* Map
* Set
* String
* TypedArray
* 函数的 arguments 对象
* NodeList 对象

### **generator**

generator实现promise

# css&动画

### **static relative absolute fixed**

* static（静态定位）：默认值。没有定位，元素出现在正常的流中（忽略 top, bottom, left, right 或者 z-index 声明）。
* relative（相对定位）：生成相对定位的元素，通过top,bottom,left,right的设置相对于其正常（原先本身）位置进行定位。可通过z-index进行层次分级。
* absolute（绝对定位）：生成绝对定位的元素，相对于 static 定位以外的第一个父元素进行定位。元素的位置通过 "left", "top", "right" 以及 "bottom" 属性进行规定。可通过z-index进行层次分级。
* fixed（固定定位）：生成绝对定位的元素，相对于浏览器窗口进行定位。元素的位置通过 "left", "top", "right" 以及 "bottom" 属性进行规定。可通过z-index进行层次分级。

### **1px问题**

这里主要有一个逻辑像素和物理像素的概念在里面

逻辑像素也就是我们常说的px

物理像素就是分辨率，单位是pt，它表示手机再垂直和水平上所具有的像素点数，一般一个pt为0.376mm

而物理像素与逻辑像素的比值被称为DPR（设备像素比），以 iPhone7 为例子，iPhone7 的 DPR = iPhone7 物理像素 / iPhone7 设备独立像素 = 2

所以设计同学给到的尺寸通常为物理像素，而我们使用px确实逻辑像素。物理像素的1，对应在前端开发中也许就是0.5px。而安卓不支持直接使用0.5px做展示，所以需要特殊处理，一般方式是使用伪元素+trasform:scale(0.5)

### **em&rem**

em：相对于父元素

子元素字体大小的em是相对于父元素字体大小

元素的width/height/padding/margin用em的话是相对于该元素的font-size

rem：相对于根元素

### **CSS选择器的优先级如何计算**

选择器从高到低如下：

1. important
2. 内敛样式
3. ID选择符0,x,0,0

（3）class选择符/属性选择符/伪类选择符  0,0,x,0

（4）元素和伪选择符0,0,0,x

### **css3动画效果有哪些**

<https://segmentfault.com/a/1190000022540857>

css动画实现方式有如下几种

* transition 实现渐变动画:属性包括需要过渡的样式属性、过渡需要的时间、过渡的速度曲线、过渡延迟触发的时间
* transform 转变动画,该属性允许我们能够对元素进行旋转、缩放、倾斜、移动这四类操作.一般是配合transition的属性一起使用。
* animation 实现自定义动画：属性主要包括需要绑定的选择器的keyfrane名称、动画完成时间、动画速度函数、动画开始前的延迟时间、动画执行次数、是否轮询

### **如何做移动端适配**

在移动端视窗有三个概念：布局视窗、视觉视窗、理想视窗

从适配的目的是在屏幕大小不同的终端设备拥有统一的界面，让拥有更大屏幕的终端展示更多的内容，从上面两个角度出发，核心思路是：页面内容充满整个视窗

1. **需要做viewport适配，在meta标签设置viewport**

<meta name="viewport" content="width=device-width,initial-scale=1">

1. **rem 布局**

rem 是CSS3新增的一个相对单位，它以 HTML 元素的 font-size 为比例。这样可以让视窗按照一定比例分成一分一分的，让内容充满视窗且无横向滚动条

缺点：在大屏设备（Pad）上，元素尺寸会很大，页面显示更少的内容。需要限制rem的最大值

1. **vw/vh**

优点：vw、vh布局能良好的实现在不同尺寸的屏幕横向填满屏幕，使用 postcss-px-to-viewport 能很好的帮我们进行单位转换

缺点：

1. 无法修改 vw/vh 的值，在大屏设备（Pad）中元素会放大，且无法通过 js 干预
2. 兼容性- 大多数浏览器都支持、ie11不支持 少数低版本手机系统 ios8、android4.4以下不支持
3. **使用媒体查询**

通过媒体查询，可以针对不同的屏幕进行单独设置，但是针对所有的屏幕尺寸做适配显然是不合理的，但是可以用来处理极端情况（例如 IPad 大屏设备）或做简单的适配（隐藏元素或改变元素位置）

**综合考量**

1. 编写 <meta> 标签设置 viewport 的内容 width=device-width，让网页宽度等于视窗宽度
2. 在 css 中使用 px
3. 在适当的场景使用flex布局，或者配合vw进行自适应
4. 在跨设备类型的时候（pc <-> 手机 <-> 平板）使用媒体查询
5. 在跨设备类型如果交互差异太大的情况，考虑分开项目开发

### **css预处理器的优缺点**

**什么是CSS预处理?**

CSS 本身不属于可编程语言，当前端项目逐渐庞大之后 CSS 的维护也愈加困难。CSS 预处理器所做的本质上是为 CSS 增加一些可编程的特性，通过变量、嵌套、简单的程序逻辑、计算、函数等特性，通过工程化的手段让 CSS 更易维护，提升开发效率。

**sass常用的几种功能：**

* 变量：变量中可以存储颜色、字体或任何 CSS 值。
* 嵌套：可嵌套 CSS 选择器，提供清晰的层次结构。
* 混合：可以定义&重用代码块。
* 扩展/集成：可以在一个选择器内继另一个选择器。
* 操作符：可以在 CSS 中使用操作符进行计算。
* 条件/循环语句：可以循环/条件生成 CSS。
* 自定义函数：可以自定义复杂操作的函数。

**优点：**

* 使用广泛。
* 功能支持完善。
* 可编程能力强。

**缺点：**

复杂度不可控、有一定的学习门槛

### **less的&用法**

* 用于伪类
* 用于内层选择器，作为对父选择器的引用

### **清除浮动有哪些方法**

* 空div方法：在需要清除浮动的元素后面添加一个空白标签<div style="clear:both;"></div>
* 父级元素添加overflow: auto或overflow: hidden方法，使用BFC
* 伪元素清除浮动，最好的方法
* 父元素设置display: table

### **BFC是什么吗 怎么触发BFC**

块级格式上下文，简单来说bfc就相当于一个大盒子，区域内的一切元素怎样设置都不会影响外部元素。触发BFC可以清除浮动、让margin不重叠。

触发条件:

* overflow的值不为visible
* display的值为flex、grid、table-cell、table-caption和inline-block之一。
* position的值不为static或relative中的任何一个
* float不为 none
* 一个新的 display 属性的值，它可以创建无副作用的 BFC。在父级块中使用 display: flow-root 可以创建新的 BFC。

BFC能解决：

* margin 重合问题
* margin 塌陷问题
* 高度塌陷问题

### **盒子模型的理解**

* 标准盒(content-box): 设置的宽高只是包括内容区。
* E盒(border-box):设置的宽高包含了内边距和边框。

### **svg和canvas的区别**

* canvas绘制位图,绘制出来的每一个图形的元素都是独立的DOM节点，能够方便的绑定事件或用来修改。canvas复杂度高会减慢渲染速度（任何过度使用 DOM 的应用都不快）。canvas输出的是一整幅画布，就像一张图片一样，放大会失真。canvas不适合游戏应用。
* svg输出的图形是矢量图形，后期可以修改参数来自由放大缩小，SVG 图像在放大或改变尺寸的情况下其图形质量不会有所损失。svg最适合图像密集型的游戏，其中的许多对象会被频繁重绘

### **浏览器如何解析css选择器**

CSS选择器的解析是从右向左解析的。若从左向右的匹配，发现不符合规则，需要进行回溯，会损失很多性能。若从右向左匹配，先找到所有的最右节点，对于每一个节点，向上寻找其父节点直到找到根元素或满足条件的匹配规则，则结束这个分支的遍历。两种匹配规则的性能差别很大，是因为从右向左的匹配在第一步就筛选掉了大量的不符合条件的最右节点（叶子节点），而从左向右的匹配规则的性能都浪费在了失败的查找上面。

而在 CSS 解析完毕后，需要将解析的结果与 DOM Tree 的内容一起进行分析建立一棵 Render Tree，最终用来进行绘图。在建立 Render Tree 时（WebKit 中的「Attachment」过程），浏览器就要为每个 DOM Tree 中的元素根据 CSS 的解析结果（Style Rules）来确定生成怎样的 Render Tree。

### **文字截断**

### **7、隐藏元素的方法**

display:none;opacity:0;visiblity:hidden;

### **8、flex**



### **9、垂直居中**

1、用flex 父：display:flex;justify-content:center;align-items:center;

2、用table-cell 父：display:table-cell;vertical-align:middle;text-align:center;

3、用margin的 auto+transform 子：margin:0 auto;transform:translateY(50%);

4、用行内块元素加行高 父：height:100px;line-height:100px; 子：display:inline-block;

5、absolute+margin 子：position:absolute;left:0;top:0;right:0;bottom:0;margin:auto

6、absolute+transform 子：position:absolute;left:50%;top:50%;transform:translate(-50%,-50%)

7、网格布局 父：display:grid 子：align-self:center;junstify-self:center  
width 、 max-width 、min-width优先级

### **10、min-width>width和max-width和flex-basis**

当三个属性都设置时，相当于取Math.max(min-width, Math.min(width, max-width))，

flex-basis的不允许小于0，小于0的话这个属性就相当于不存在，所以都是随着width走。

当flex-basis>=0时，如果flex-basis小于自身内容宽度，则如果 width也设置了，那么flex-basis和width哪个值大就取哪个值。flex-basis<0时则为width的宽度。

### **11、png、jpg、gif 这些图片格式解释一下，分别什么时候用。有没有了解过webp？**

png是便携式网络图片（Portable Network Graphics）是一种无损数据压缩位图文件格式.优点是：压缩比高，色彩好。 大多数地方都可以用。

jpg是一种针对相片使用的一种失真压缩方法，是一种破坏性的压缩，在色调及颜色平滑变化做的不错。在www上，被用来储存和传输照片的格式。

gif是一种位图文件格式，以8位色重现真色彩的图像。可以实现动画效果.

webp格式是谷歌在2010年推出的图片格式，压缩率只有jpg的2/3，大小比png小了45%。缺点是压缩的时间更久了，兼容性不好，目前谷歌和opera支持。

# 浏览器相关

### **浏览器内核有哪几种**

**什么是浏览器内核？**

1、浏览器内核也可以理解为排版引擎，主要由2部分组成，渲染引擎和 JS 引擎

2、排版引擎主要是负责获取网页内容（HTML、XML、图像、CSS、JS）等信息进行解析、渲染网页，将网页的代码转换为看得到的页面

3、浏览器的内核的不同对于网页的语法解释会有不同，所以渲染的效果也不相同，常见的内核有Trident、 Gecko、 Webkit、 Presto、 Blink五种

**Trident**

代表作品是IE，俗称IE内核；使用IE内核的浏览器包括、傲游、世界之窗、百度浏览器、兼容模式的浏览器等

**WebKit内核**

代表作品是Safari、旧版的Chrome

**Presto内核**

代表作品是Opera，Presto是由Opera Software开发的浏览器排版引擎，它是世界公认最快的渲染速度的引擎。在13年之后，Opera宣布加入谷歌阵营，弃用了 Presto

**Blink内核**

代表作品是Chrome、Opera；由Google和Opera Software开发的浏览器排版引擎

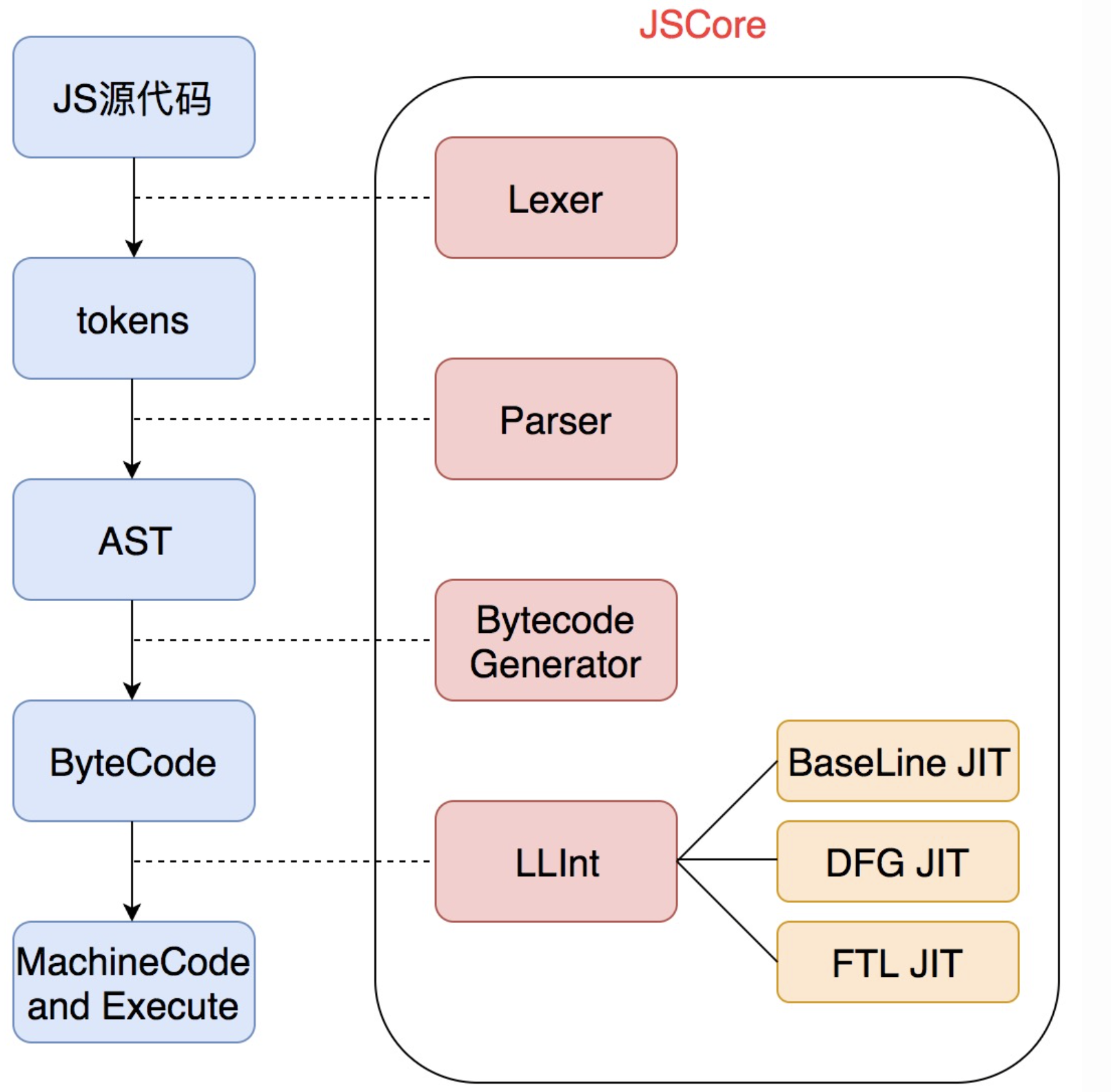
**Gecko内核**

代表作品是Firefox，俗称Firefox内核

**Chromium内核**

这个比较特殊，Chromium是谷歌的开源项目是一款浏览器，Chrome 是Chromium的稳定版。国内的大部分双核浏览器都采用Chromium内核

### **js引擎的工作过程**



js代码语法解析为令牌流，令牌流再解析为抽象语法树，之后抽象语法树解析为字节码，拿到字节码以后会删除抽象语法树，生成的字节码会进行缓存

### **浏览器的主要组成部分**

* 用户界面 ：地址栏，后退/前进按钮，书签目录等；
* 浏览器引擎：用户界面和渲染引擎之间传送指令
* 渲染引擎：负责显示请求的内容
* 网络：用于网络调用，比如 HTTP 请求
* 用户界面后端： 用来绘制类似组合选择框及对话框等基本组件
* JavaScript解释器：用于解析和执行 JavaScript 代码
* 数据存储：浏览器内数据库

### **浏览器的本地存储，各自的优缺点如何**

* cookie: 为例弥补HTTP在状态管理上的不足。向同一个域名下发送请求，都会携带相同的 Cookie，服务器拿到 Cookie 进行解析，便能拿到客户端的状态。但是会有一系列的缺点
  + 容量缺陷：Cookie 的体积上限只有4KB
  + 性能缺陷：不管域名下面的某一个地址需不需要这个 Cookie ，请求都会携带上完整的 Cookie，请求携带了不必要的内容
  + 安全缺陷：Cookie 以纯文本的形式在浏览器和服务器中传递，很容易被非法用户截获，然后进行一系列的篡改
* localStorage：
  + 容量。localStorage 的容量上限为5M，相比于 Cookie 的 4K 大大增加。当然这个 5M 是针对一个域名的，因此对于一个域名是持久存储的。
  + 只存在客户端，默认不参与与服务端的通信。这样就很好地避免了 Cookie 带来的性能问题和安全问题。
  + 接口封装。通过 localStorage 暴露在全局，并通过它的 setItem 和 getItem等方法进行操作，非常方便。
* sessionStroage：sessionStorage 和 localStorage 有一个本质的区别，那就是前者只是会话级别的存储，并不是持久化存储。会话结束，也就是页面关闭，这部分 sessionStorage 就不复存在了。
* indexdb：容量50m

### **dom树如何构建的？**

1. 转码: 浏览器将接收到的二进制数据按照指定编码格式转化为HTML字符串
2. 生成Tokens: 之后开始parser，浏览器会将HTML字符串解析成Tokens
3. 构建Nodes: 对Node添加特定的属性，通过指针确定 Node 的父、子、兄弟关系和所属 treeScope
4. 生成DOM Tree: 通过node包含的指针确定的关系构建出DOM Tree、

### **浏览器是如何渲染ui的**

1. 浏览器获取html文件并解析，生成dom树
2. 同时进行css解析，生成样式规则。然后和dom树结合为渲染树
3. 渲染树进行布局处理，生成布局树，也就是每个节点分配一个应出现的屏幕上的确切的坐标
4. 随后调用gup进行绘制，遍历树节点将元素呈现出来

### **什么是重绘和回流**

**什么是重绘?**

重绘: 改变某个元素的背景色、文字颜色、边框颜色等等，而这些属性不会影响元素的布局，我们将这样的操作称为重绘。由于此时只需要进行渲染过程的最后一步，重新绘制，即UI层面的绘制，因此损耗较小

**什么是回流**

当渲染树中的一部分（或全部）因为元素的规模尺寸、布局、隐藏等改变而需要重新构建的操作，会影响到布局的操作，需要重新验证计算渲染树，这样的操作我们称为回流。

（1）添加或者删除可见的 DOM 元素；

（2）元素尺寸改变——边距、填充、边框、宽度和高度

（3）内容变化，比如用户在 input 框中输入文字

（4）浏览器窗口尺寸改变——resize事件发生时

（5）计算 offsetWidth 和 offsetHeight 属性

（6）设置 style 属性的值

（7）当你修改网页的默认字体时。

### **如何避免重绘和回流**

**CSS:**

* 避免使用table布局
* 尽可能再dom树的末端修改class
* 避免使用多层内联样式
* 将动画效果应用到position: absolute || fixed上
* 避免使用css表达式（例如calc）
* CSS3硬件加速（GPU加速）

**JavaScript:**

* 避免频繁操作样式，最好一次性修改style属性，或者将样式列表定义成class，并一次性更改class属性
* 避免频繁操作dom，创建一个documentFragment，在他上面应用所有的dom操作，最后再把他添加到文档中
* 也可以先为元素设置display: none，操作结束后再把它显示出来，因为再display为none的元素上进行dom操作不会引发重绘和回流
* 避免频繁读取会引发重绘回流的属性，如果需要多次使用，就用一个变量缓存起来
* 对具有复杂动画的元素使用绝对定位，使他脱离文档流，否则会引起父元素及后续元素频繁回流
* 使用cssText来更改样式

### **前端如何实现即时通信**

**短轮询**

短轮询的原理很简单，每隔一段时间客户端就发出一个请求，去获取服务器最新的数据

* 优点：兼容性强，实现非常简单
* 缺点：延迟性高，非常消耗请求资源，影响性能

**长轮询和流**

comet 是一种高级的ajax技术，有两种主要实现手段，长轮询和http流。

长轮询是页面向服务器发送一次请求，tcp连接一直不断开，直到有数据响应以后，发送完数据后，该请求被关闭，随即页面又发起一个新的服务器请求，这个过程一直循环

http流，在页面的整个生命周期都使用一个http连接，具体使用方法就是浏览器向服务端发起请求，服务器一直保持tcp连接打开，然后不断向浏览器发送数据

* 优点：兼容性好，资源浪费较小
* 缺点：服务器hold连接会消耗资源，返回数据顺序无保证，难于管理维护

**websocket**

Websocket 是一个全新的、独立的协议，基于 TCP协议，与 http协议 兼容、却不会融入http协议，仅仅作为html5的一部分，其作用就是在服务器和客户端之间建立实时的双向通信。

* 优点：真正意义上的实时双向通信，性能好，低延迟
* 缺点：独立与http的协议，因此需要额外的项目改造，使用复杂度高，必须引入成熟的库，无法兼容低版本浏览器

**SSE**

SSE（Server-Sent Event，服务端推送事件）是一种允许服务端向客户端推送新数据的HTML5技术，是对comet交互推出的api

* 优点：基于HTTP而生，因此不需要太多改造就能使用，使用方便，而websocket非常复杂，必须借助成熟的库或框架
* 缺点：基于文本传输效率没有websocket高，不是严格的双向通信，客户端向服务端发送请求无法复用之前的连接，需要重新发出独立的请求

### **websocket的特点**

1. 建立在tcp之上
2. 很好的兼容了http协议。默认端口是80和443。未加密的ur为ws、加密的url为wss。并且握手阶段采用http协议，所以不容易被屏蔽。
3. 数据格式比较轻量级，性能开销也小，通信高效
4. 可以发送文本也可以发送二进制数据
5. 没有同源的限制，可以与任意服务器通信

### **前端实现跨域**

**浏览器同源策略**

“协议+域名(host)+端口”三者相同

* 一级域名相同二级域名不同不算跨域
* 移动端的webview并 不存在跨域问题，因为webview是通过file://协议来加载html文件的h

**不受同源限制的三个标签**

* <img src=XXX>
* <link href=XXX>
* <script src=XXX>

**同源政策主要限制了三个方面**

* 第一个是当前域下的 js 脚本不能够访问其他域下的 cookie、localStorage 和 indexDB。
* 第二个是当前域下的 js 脚本不能够操作访问其他域下的 DOM。
* 第三个是当前域下 ajax 无法发送跨域请求。

同源政策的目的主要是为了保证用户的信息安全，它只是对 js 脚本的一种限制，并不是对浏览器的限制，对于一般的 img、或者 script 脚本请求都不会有跨域的限制，这是因为这些操作都不会通过响应结果来进行可能出现安全问题的操作。

**跨域场景**

<https://blog.csdn.net/haotian1997/article/details/120006209>

* **document.domain**
* **ifram+window.name**
* **ifram+window.open**
* **jsonp**
* **cors**
* **nginx**
* **websocket**
* **postMessage**
* **node中间件代理**

#### 9.0 document.domain

这种方式不适用于cookie

如果父域相同，比如b.a.com与 a.com可通过通过使用document.domain或者cookie来将b.a.com修改为a.com来实现跨域。

#### 9.1 jsonp

实现JSONP跨域请求还需要服务器端配合接口，动态生成一个js代码并将要返回的数据封装在返回的js脚本中如上面的test.js文件，这时请求端还应提前定义好被调用的函数（回调函数），以及在请求上加入参数，且这个参数是一个回调函数

实现jsonp

#### 9.2 cors(跨域资源共享)

cors 需要浏览器和后端同时支持，IE 8 和 9 需要通过 XDomainRequest 实现

cors 分为简单请求和非简单请求

1. 简单请求

满足两大条件:

1. 使用下列方法: GET、POST、HEAD 之一
2. Content-Type值权限于下列三者之一: text/plain、multipart/form-data、application/x-www-form-urlencoded

简单请求实现跨域就是设置Access-Control-Allow-Origin

1. 非简单请求

不符合以上条件就是复杂请求了。复杂请求的跨域实现是通过 Options 方法对指定url发送嗅探请求，判断是否有权限访问

服务端设置 Access-Control-Allow-Origin 就可以开启 CORS。 该属性表示哪些域名可以访问资源，如果设置通配符则表示所有网站都可以访问资源。

* Access-Control-Allow-Origin：设置哪个源可以访问我
* Access-Control-Allow-Methods：允许携带哪个方法访问我
* Access-Control-Allow-Headers：允许携带哪个头访问我
* Access-Control-Allow-Credentials：允许携带cookie
* Access-Control-Max-Age：预计的存活时间

#### 9.3 websocket

WebSocket 是一种双向通信协议，在建立连接之后，WebSocket 的 server 与 client 都能主动向对方发送或接收数据，连接建立好了之后 client 与 server 之间的双向通信就与 HTTP 无关了，因此可以跨域。

原生WebSocket API使用起来不太方便，我们使用 Socket.io，它很好地封装了webSocket接口，提供了更简单、灵活的接口，也对不支持webSocket的浏览器提供了向下兼容



#### 9.4 postMessage()方法

HTML5 XMLHttpRequest 有一个API，postMessage() ，且是为数不多可以跨域操作的window属性之一，它可用于解决以下方面的问题

* 页面和其打开的新窗口的数据传递
* 多窗口之间消息传递
* 页面与嵌套的iframe消息传递
* 上面三个场景的跨域数据传递

postMessage() 方法允许来自不同源的脚本采用异步方式进行有限的通信，可以实现跨文本档、多窗口、跨域消息传递

#### 9.5 node中间代理

实现原理：同源策略是浏览器需要遵循的标准，而如果是服务器向服务器请求就无需遵循同源策略。 代理服务器

### **输入url发生了什么**

**加载过程**

* URL解析，如果有非法字符，就转义
* 浏览器查找当前URL是否存在缓存，并比较缓存是否过期
* DNS解析域名,域名->IP地址
* 浏览器与服务器建立tcp链接（三次握手）
* 发送请求，分析url，设置请求头
* 服务器返回请求的文件（html）

**渲染过程**

* 根据HTML、Css代码生成相应的DOMTree,Style Rules
* 结合DOMTree和Style Rules生成RenderTree，然后将css挂载在DOM上
* 根据RenderTree渲染页面
* 页面遇到<script></script>标签停止渲染,执行完js代码后再继续渲染
* layout布局渲染
* GPU像素绘制页面
* 直至渲染完成

**可能会追问的问题**

* 先说为什么url要解析（也就是编码)

因为网络标准规定了URL只能是字母和数字，还有一些其它特殊符号,如果不转义就会出现歧义，比如key=value，此时key就是=字符，就会出现歧义

* url编码的规则是什么呢

utf-8，中文的话用gb2312。浏览器通过 encodeURIComponent 统一编码格式

* encodeURIComponent 和 encodeURI有什么区别

区别就是 encodeURIComponent 编码范围更广，适合给参数编码，encodeURI 适合给URL本身（locaion.origin）编码,当然项目里一般都是用qs库去处理

### **渲染线程和js引擎线程**

浏览器中常见的线程有：渲染线程、JS 引擎线程、HTTP 线程等等。

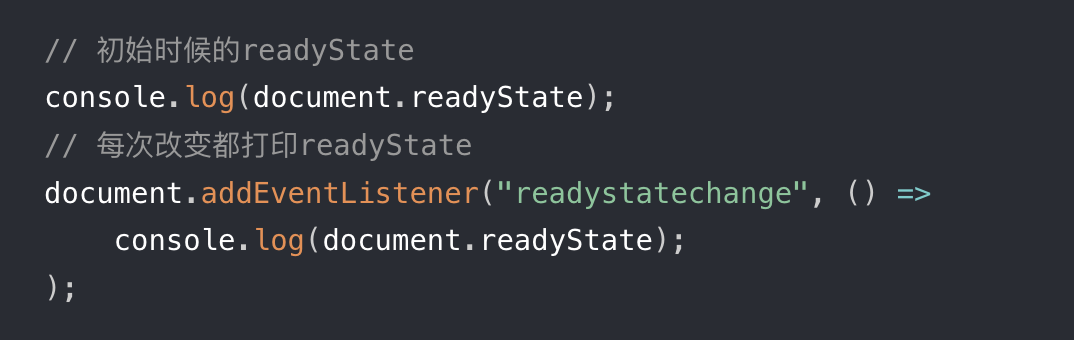
例如，当我们打开一个 Ajax 请求的时候，就启动了一个 HTTP 线程。

同样地，我们可以用线程的只是解释：为什么直接操作 DOM 会变慢，性能损耗更大？因为 JS 引擎线程和渲染线程是互斥的。而直接操作 DOM 就会涉及到两个线程互斥之间的通信，所以开销更大。

除此之外，这还能解释为什么 <script> 标签为什么会阻塞 DOM 树渲染，毕竟 JS 是可以修改 DOM 的，如果 JS 执行的时候 UI 也工作，就有可能导致不安全的渲染。

### **页面生命周期**

页面声明周期的变化，会触发document上的readystatechange事件，用户可以通过document.readyState拿到当前的状态。



上面的代码在 Chrome 中的输出是：

* loading：加载 document
* interactive：document 加载成功，DOM 树构建完成
* complete：图像，样式表和框架之类的子资源完成加载

**onload和DOMContentLoaded触发的先后顺序是什么？**

【生命周期】：

* DOMContentLoaded事件在 DOM 树构建完毕后被触发，我们可以在这个阶段使用 js 去访问元素。
* async和defer的脚本可能还没有执行。
* 图片及其他资源文件可能还在下载中。
* load事件在页面所有资源被加载完毕后触发，通常我们不会用到这个事件，因为我们不需要等那么久。
* beforeunload在用户即将离开页面时触发，它返回一个字符串，浏览器会向用户展示并询问这个字符串以确定是否离开。

### **cookie、session、sessionStorage、cookieStorage**

Cookie 和 session 都可用来存储用户信息，cookie 存放于客户端，session 存放于服务器端，因为 cookie 存放于客户端有可能被窃取，所以 cookie 一般用来存放不敏感 的信息，比如用户设置的网站主题，敏感的信息用 session 存储，比如用户的登陆信息，session 可以存放于文件，数据库+，内存中都可以，cookie 会在请求时在 http 首部发送给客户端，

下面从几个方向区分一下 cookie，localstorage，sessionstorage 的区别

* 生命周期:

Cookie:可设置失效时间，否则默认为关闭浏览器后失效 Localstorage:除非被手动清除，否则永久保存 Sessionstorage:仅在当前网页会话下有效，关闭页面或浏览器后就会被清除

* 存放数据:

Cookie:4k 左右

Localstorage 和 sessionstorage:可以保存 5M 的信息

* http 请求:

Cookie:每次都会携带在 http 头中，如果使用 cookie 保存过多数据会带来性能问题 其他两个:仅在客户端即浏览器中保存，不参与和服务器的通信

* 易用性:

Cookie:需要程序员自己封装，原生的 cookie 接口不友好 其他两个:即可采用原生接口，亦可再次封装

* 应用场景:

从安全性来说，因为每次 http 请求都回携带 cookie 信息，这样子浪费了带宽，所以 cookie 应该尽可能的少用，此外 cookie 还需要指定作用域，不可以跨域调用，限制很 多，但是用户识别用户登陆来说，cookie 还是比 storage 好用，其他情况下可以用 storage，localstorage 可以用来在页面传递参数，sessionstorage 可以用来保存一 些临时的数据，防止用户刷新页面后丢失了一些参数。

### **ajax工作原理及优缺点**

AJAX即“Asynchronous Javascript And XML”，Ajax是一种异步请求数据的web开发技术，对于改善用户的体验和页面性能很有帮助。简单地说，在不需要重新刷新页面的情况下，Ajax 通过异步请求加载后台数据，并在网页上呈现出来。常见运用场景有表单验证是否登入成功、百度搜索下拉框提示和快递单号查询等等。

**原理：**

**其中最核心的依赖是浏览器提供的XMLHttpRequest对象，**客户端发送请求，请求交给xhr，xhr把请求提交给服务，服务器进行业务处理，服务器响应数据交给xhr对象，xhr对象接收数据，由javascript把数据写到页面上，如下图所示：

* 创建 Ajax的核心对象 XMLHttpRequest对象
* 通过 XMLHttpRequest 对象的 open() 方法与服务端建立连接
* 构建请求所需的数据内容，并通过XMLHttpRequest 对象的 send() 方法发送给服务器端
* 通过 XMLHttpRequest 对象提供的 onreadystatechange 事件监听服务器端你的通信状态
* 接受并处理服务端向客户端响应的数据结果
* 将处理结果更新到 HTML页面中

**优点**

1. 在不刷新页面的情况下与服务端通信
2. 用异步的方式与服务端通信，不阻碍用户的操作
3. 实现界面与应用分离，即数据与呈现分离

**缺点**

1. 无法用history和back机制来实现通过ajax更新的页面的回退
2. 不利于搜索引擎优化，一般搜不到ajax添加到页面的信息

### **前端异常捕获处理方式**

<https://blog.csdn.net/weixin_27283495/article/details/112224936>

**常见的需要捕获的异常**

* JS 语法错误、代码异常：try cath + window.onerror
* AJAX 请求异常：
* 静态资源加载异常：window.addEventListener('error',(){})
* Promise 异常：window.addEventListener('unhandlerejection',(){})
* Iframe 异常:window.onerror
* 跨域 Script error：script.onerror+标签设置crossorigin='anonymous'
* 崩溃和卡顿：1）load+beforeunload; 2)serviceworker

# 框架相关

## webpack

### **1、webpack和glup的区别**

gulp 强调的是前端开发的工作流程，我们可以通过配置一系列的 task，定义 task 处理的事务(例如文件压缩合并、雪碧图、启动 server、版本控制等)，然后定义执行 顺序，来让 gulp 执行这些 task，从而构建项目的整个前端开发流程。

webpack 是一个前端模块化方案，更侧重模块打包，我们可以把开发中的所有资源(图 片、js 文件、css 文件等)都看成模块，通过 loader(加载器)和 plugins(插件) 对资源进行处理，打包成符合生产环境部署的前端资源。

### **2、webpack工作流程**

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

* 初始化参数：从配置文件和 Shell 语句中读取与合并参数，得出最终的参数；
* 开始编译：用上一步得到的参数初始化 Compiler 对象，加载所有配置的插件，执行对象的 run 方法开始执行编译；
* 确定入口：根据配置中的 entry 找出所有的入口文件；
* 编译模块：从入口文件出发，调用所有配置的 Loader 对模块进行翻译，再找出该模块依赖的模块，再递归本步骤直到所有入口依赖的文件都经过了本步骤的处理；

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

* 输出资源：根据入口和模块之间的依赖关系，组装成一个个包含多个模块的 Chunk，再把每个 Chunk 转换成一个单独的文件加入到输出列表，这步是可以修改输出内容的最后机会；
* 输出完成：在确定好输出内容后，根据配置确定输出的路径和文件名，把文件内容写入到文件系统。

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

### **3、webpack热更新流程**

主要依赖 webpack, express, websocket

* 使用 express 开启本地服务，服务端和客户端使用 websocket 开启长连接
* webpack 监听源文件的变化：每次编译完成之后都会生成: hash值，改动模块的json文件,改动模块的js文件，然后向客户端推送当前编译的哈希值
* 客户端拿到哈希值后，会和上一次进行比较，如果一致就拿缓存，如果不一致就通过 ajax 和 jsonp 获取最新的资源
* 使用内存文件系统替换修改内容实现更新

### **4、webpack的proxy是如何解决跨域的?**

我们在使用webpack开发项目的时候，webpack的dev-server模块会启动一个服务器，这个服务器不止帮我们做了自动更新，同时也可以做到反向代理。 就是我们把请求发送给webpack-dev-server, 然后webpack-dev-server再去请求后端服务器，服务之间的请求是没有跨域问题的，只要后端返回了webpack-dev-server就能拿到，然后再返回给前端。

### **5、loader**

#### 5.1 基本概念

Loader 就像是一个翻译员，能把源文件经过转化后输出新的结果，并且一个文件还可以链式的经过多个翻译员翻译。

一个 Loader 的职责是单一的，只需要完成一种转换。 如果一个源文件需要经历多步转换才能正常使用，就通过多个 Loader 去转换。 在调用多个 Loader 去转换一个文件时，每个 Loader 会链式的顺序执行， 第一个 Loader 将会拿到需处理的原内容，上一个 Loader 处理后的结果会传给下一个接着处理，最后的 Loader 将处理后的最终结果返回给 Webpack。

所以，在你开发一个 Loader 时，请保持其职责的单一性，你只需关心输入和输出

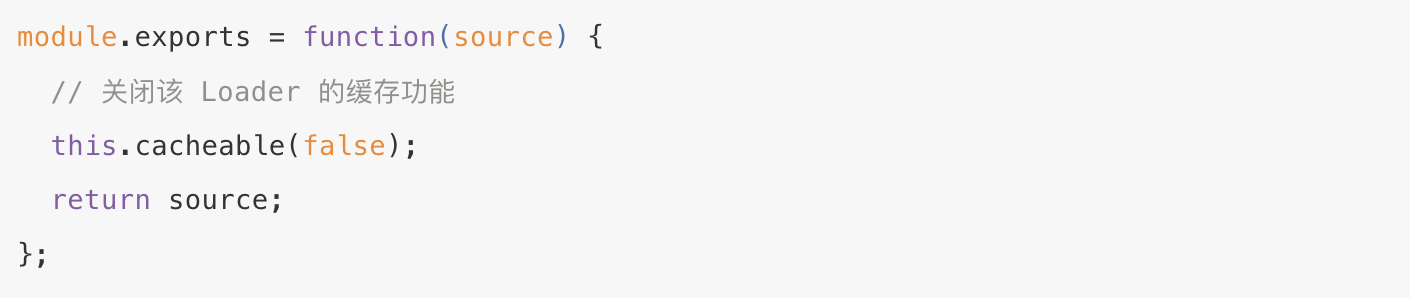
以处理 SCSS 文件为例：

* SCSS 源代码会先交给 sass-loader 把 SCSS 转换成 CSS；
* 把 sass-loader 输出的 CSS 交给 css-loader 处理，找出 CSS 中依赖的资源、压缩 CSS 等；
* 把 css-loader 输出的 CSS 交给 style-loader 处理，转换成通过脚本加载的 JavaScript 代码；

#### 5.2 laoder缓存加速

在有些情况下，有些转换操作需要大量计算非常耗时，如果每次构建都重新执行重复的转换操作，构建将会变得非常缓慢。 为此，Webpack 会默认缓存所有 Loader 的处理结果，也就是说在需要被处理的文件或者其依赖的文件没有发生变化时， 是不会重新调用对应的 Loader 去执行转换操作的。

如果你想让 Webpack 不缓存该 Loader 的处理结果，可以这样：



#### 5.3 加载本地laoder

**npm link**

Npm link 专门用于开发和调试本地 Npm 模块，能做到在不发布模块的情况下，把本地的一个正在开发的模块的源码链接到项目的 node\_modules 目录下，让项目可以直接使用本地的 Npm 模块。 由于是通过软链接的方式实现的，编辑了本地的 Npm 模块代码，在项目中也能使用到编辑后的代码。

**ResolveLoader**

 ResolveLoader 用于配置 Webpack 如何寻找 Loader。 默认情况下只会去 node\_modules 目录下寻找，为了让 Webpack 加载放在本地项目中的 Loader 需要修改 resolveLoader.modules。

### **6、pulgin**

Webpack 通过 Plugin 机制让其更加灵活，以适应各种应用场景。Webpack 通过 [Tapable](https://github.com/webpack/tapable) 来组织这条复杂的生产线。 在 Webpack 运行的生命周期中会广播出许多事件，Plugin 可以监听这些事件，在合适的时机通过 Webpack 提供的 API 改变输出结果。

webpack 插件是一个具有 apply方法的 JavaScript 对象。

apply方法中会传入compiler对象，compiler对象中含有各个生命周期的hooks，我们按照自己的需求监听对应做处理

**常用的生命周期钩子有哪些**

由于webpack的流程主要三个阶段【初始化】=》【编译】=》【输出】，

其中**初始化阶段**比如environment，一个环境准备好的阶段。

**编译阶段：**比如compiler：即将开始一次新的构建、compilation：新的构建开始、make：compilation创建完成，即将从entry开始读取文件、after-compile：一次compilation执行完成、进行新一轮构建编译，invalid：文件编译异常。

**输出阶段**：should-emit：需要输出的文件已经生成了，询问插件哪些需要输出，emit：执行文件输出，after-emit：文件输出完成。done：完成一次编译输出流程

### **7、compiler和compilation区别**

Compiler 和 Compilation 的区别在于：Compiler 代表了整个 Webpack 从启动到关闭的生命周期，而 Compilation 只是代表了一次新的编译。

* Compiler 对象包含了 Webpack 环境所有的的配置信息，包含 options，loaders，plugins 这些信息，这个对象在 Webpack 启动时候被实例化，它是全局唯一的，可以简单地把它理解为 Webpack 实例；
* Compilation 对象包含了当前的模块资源、编译生成资源、变化的文件等。当 Webpack 以开发模式运行时，每当检测到一个文件变化，一次新的 Compilation 将被创建。Compilation 对象也提供了很多事件回调供插件做扩展。通过 Compilation 也能读取到 Compiler 对象。

### **8、提高webpack打包速度**

**优化loader配置**：缩小文件匹配范围（用include/exclude）、缓存loader的执行结果



**代码分割**：不做代码分割，主入口打包会包含node\_modules的文件，包就会非常的大。意味着每次代码修改都会重新加载这个文件。做代码分割主要使用CommonsChunkPlugin，将node-modules打成common包，将第三方库打成vender包，跟业务包分开

### **压缩代码**：使用内置的TerserWebpackPlugin，切启动多进程压缩模式，提示打包速度、ParallelUglifyPlugin

<https://tsejx.github.io/webpack-guidebook/best-practice/optimization/multi-process/>



### **9、用webpack优化前端性能**

* 使用tree-shaking，剔除不需要的代码
* 对js代码做压缩：TerserPlugin
* 进行css代码压缩：css-minimizer-webpack-plugin
* html做压缩：HtmlWebpackPlugin
* 文件大小做压缩：compression-webpack-plugin
* 图片做压缩：file-loader
* 代码分离：splitChunksPlugin

### **10、babel（js编译器）的工作过程：**

1. 解析【@babel/parse、以前用@babylon】：接收代码并转换成ast

* 解析词法解析 （代码转换成令牌流：一个扁平的语法数组）
* 语法解析（令牌流转换成ast）

1. 转换【@babel/traverse @babel/types】：遍历ast，对树进行增删改查，是plugin作用的阶段
2. 生成【@babel/genertor】：进行深度遍历，将ast转换成字符串形式的代码，并且生成sourcemap

ps：除了上面说到的核心包，还有一个关键核心包是@babel/core(babel转译器本身，提供转移api)

**AST**

抽象语法树，是源代码语法结构的一种抽象表示，它以树状的形式表现编程语言的语法结构。树上的每个节点都表示代码中的一种结构

**自定义插件**

path 表示两个节点之间连接的对象，包含了域（scope）、上下文（context）等属性，也提供了insertBefore、replaceWith、remove等方法来添加、更新、移动和删除节点。



### **11、tapable**

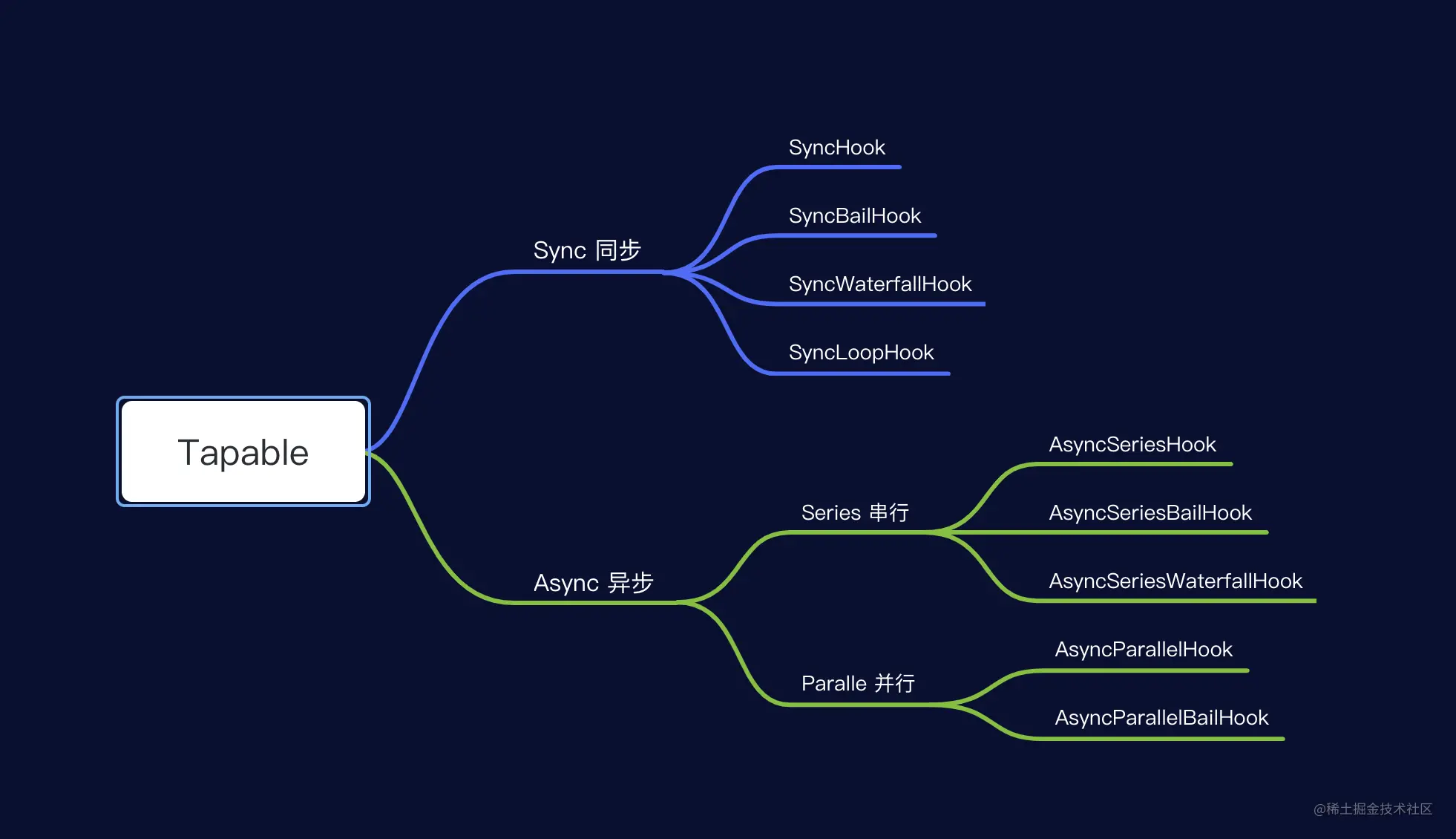
tapable实现了在webpack编译过程中的发布订阅者模式的插件机制

步骤：new初始化不同钩子 =》tap注册事件 =》 call调用事件

分类：

1. 分为同步、异步钩子

* 对于同步钩子，通过tap注册事件，call触发同步钩子的执行
* 对于异步的钩子可以通过tap，tapAsync，tapPromise三种方式注册事件，对应通过call，callAsync，callPromise来触发钩子执行



（2）按照执行机制分类

**Basic Hook**: 基本类型的钩子，它仅仅执行钩子注册的事件，并不关心每个被调用的事件函数返回值如何

**Waterfall** : 瀑布类型的钩子，瀑布类型的钩子和基本类型的钩子基本类似，唯一不同的是瀑布类型的钩子会在注册的事件执行时将事件函数执行非 undefined 的返回值传递给接下来的事件函数作为参数。

**Bail** : 保险类型钩子，保险类型钩子在基础类型钩子上增加了一种保险机制，如果任意一个注册函数执行返回非 undefined 的值，那么整个钩子执行过程会立即中断，之后注册事件函数就不会被调用了。

**Loop** : 循环类型钩子，循环类型钩子稍微比较复杂一点。循环类型钩子通过 call 调用时，如果任意一个注册的事件函数返回值非 undefeind ,那么会立即重头开始重新执行所有的注册事件函数，直到所有被注册的事件函数都返回 undefined。

### **12、Tree Shaking**

按需加载，即没有被引用的模块不会被打包进来，减少我们的包大小，缩小应用的加载时间，呈现给用户更佳的体验。

Tree Shaking实现的关键得益于ES Module模块的静态分析（Static module resolution）功能，在ES Module中，我们可以将模块的加载分为两个阶段：静态分析和编译执行；

ES Module的import语法完美可以使用tree shaking，因为可以在代码不运行的情况下就能分析出不需要的代码。

**原理**：程序会从入口文件出发扫描所有的模块依赖，以及模块的子依赖，然后将它们链接起来形成一个“抽象语法树”(AST)。随后，运行所有代码，查看哪些代码是用到过的，做好标记。最后，再将“抽象语法树”中没有用到的代码“摇落”。经历这样一个过程后，就去除了没有用到的代码。

下面我们看下ES Module的特性：

* 只能作为模块顶层的语句出现（而不嵌套在条件语句中）
* import 的模块名只能是字符串常量（只对文件进行字符串读取）
* 导入和导出语句没有动态部分（不允许使用变量等）

**实现：**

* 生产环境（当mode=production）时默认开启tree shaking
* 开发环境需要多设置一个

除此之外：

1. 为了避免所有文件被副作用影响，需要再package.json种 加入sideEffects：[.\*css]将css文件排除(默认该值为true表示所有文件都有副作用他们不能被tree shaking)
2. 避免babel将import转require，在babel的配置里面设置modules: false



### **13、\_webpack\_require**

我们在模块化开发的时候，通常会使用ES Module或者CommonJS规范导出/引入依赖模块，webpack打包编译的时候，会统一替换成自己的\_\_webpack\_require\_\_来实现模块的引入和导出，从而实现模块缓存机制，以及抹平不同模块规范之间的一些差异性。

webpack的打出来的包是一个立即执行函数

其中的核心就是 \_\_webpack\_require\_\_() 函数，它接收的参数是 moduleId，其实就是文件路径。

它的执行过程如下：

1. 判断模块是否有缓存，如果有则返回缓存模块的 export 对象，即 module.exports。
2. 新建一个模块 module，并放入缓存。
3. 执行文件路径对应的模块函数。
4. 将这个新建的模块标识为已加载。
5. 执行完模块后，返回该模块的 exports 对象。

## react

### **不能调用setstate的场景**

造成死循环：render、shouldComponentUpdate、componentWillUpdate

数据初始化还未完成：constructor

### **creactElement**

### **reactElement**

react-reconciler 负责生成Fiber树、协调和调度、产生操作指令。

而 react-dom 负责调用DOM API，将操作指令实施到DOM树上，可以将 react-dom 类比为 react-reconciler 和 DOM 之间的翻译器。

ReactElement：一个对象，包含type，props，key，ref属性，通过 children 属性描述父子关系。

### **React为什么使用虚拟dom，他解决了什么问题**

在传统网页中，需要频繁的调用dom api来更新dom结构，这样的操作非常的损耗性能，所以提出了虚拟dom的概念，每当ui状态发生变化时，先生成一个虚拟dom，虚拟dom本身是轻量级的js对象（这个对象具有批处理和高效的diff算法），这个虚拟dom与原来的dom相对比，创建并且替换有改变的节点

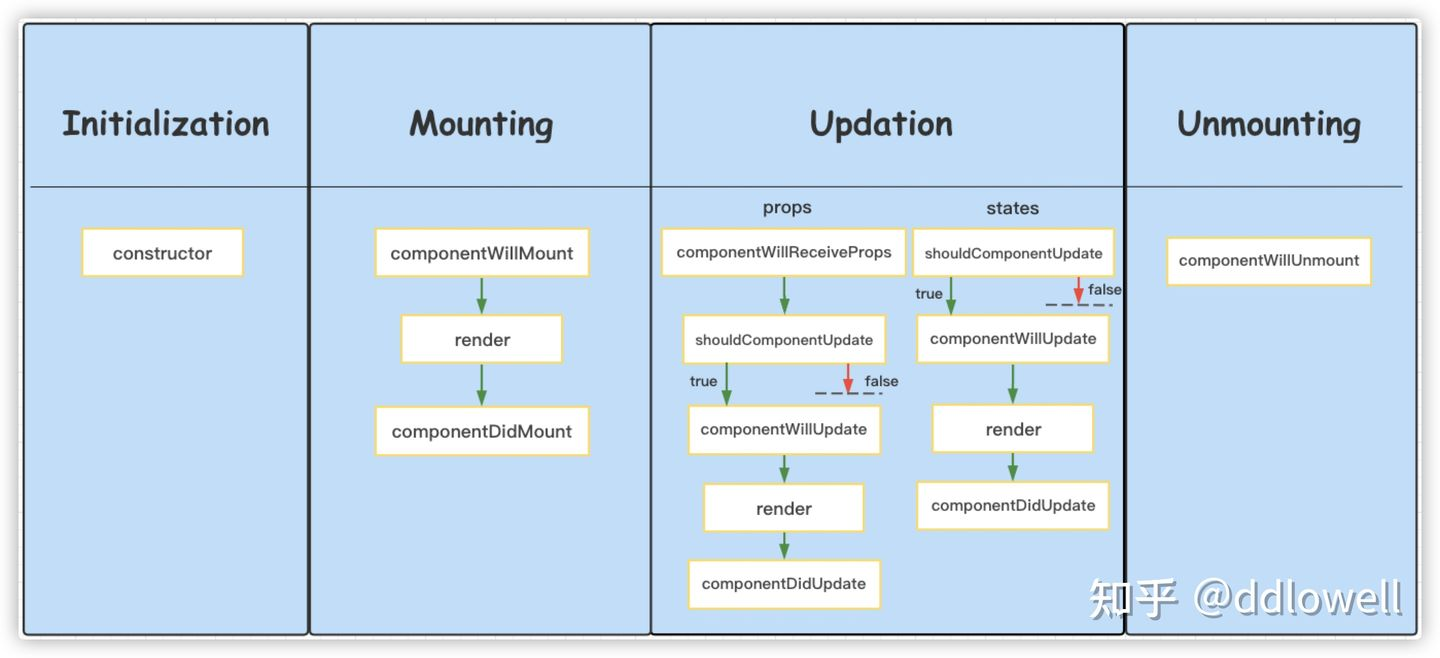
完整的树形结构的diff算法复杂度是n的三次方，但是前端很少出现跨越层级的移动dom元素，所以虚拟dom只进行同层级的元素对比，此时复杂度降到n

react更新UI的操作大致分为以下三步：

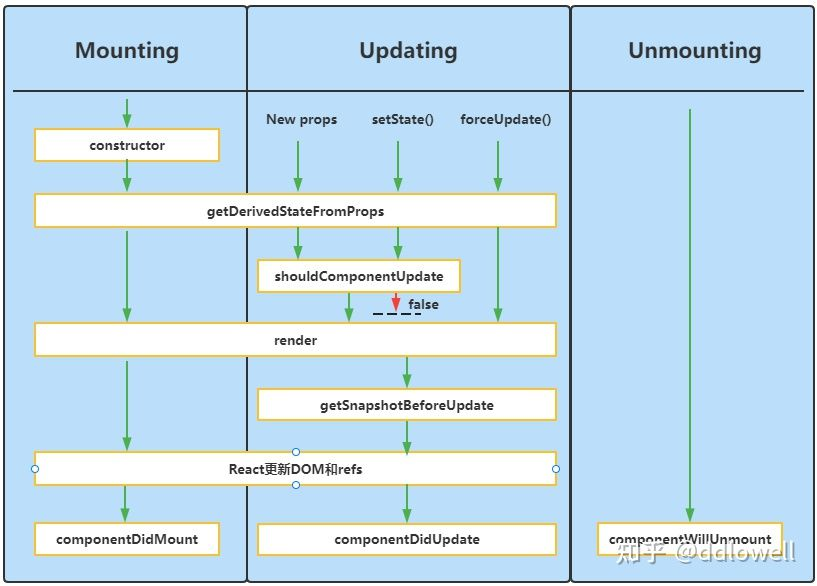
* 维护一个使用js对象表示的虚拟DOM，与真实DOM相对应。
* UI状态发生改变时，将新旧两个DOM做diff
* 更新发生变化的真实DOM

### **能完整说下React的生命周期吗（这里涉及到废弃的和新增的，以及每个生命周期的时机，以及作用）**

**react16**



**react16.3**



**getderivedstatefromprops**

getderivedstatefromprops为静态的，不能够通过this访问class的属性，作用是从prpos中获取state，必须有返回值，返回为null则表示不更新state

这个生命周期函数是为了替代componentWillReceiveProps存在的，所以在你需要使用componentWillReceiveProps的时候，就可以考虑使用getDerivedStateFromProps来进行替代了。

两者的参数是不相同的，而getDerivedStateFromProps是一个静态函数，也就是这个函数不能通过this访问到class的属性，也并不推荐直接访问属性。而是应该通过参数提供的nextProps以及prevState来进行判断，根据新传入的props来映射到state。

getSnapshotBeforeUpdate，在真实dom更新前，他的任何返回值被作为componentdidupdate的第三个参数被使用

### **Vue 和 React 有什么不同?使用场景分别是什么**

* 组件通信的区别：react的数据流是单向的，vue的数据是双向绑定的
* 响应方式不同：vue数据的改变会直接反应到视图上，而react是需要调用setState通知更新
* 渲染方式不同:react用的是jsx渲染模板，vue是一种拓展的html语法来进行渲染
* vue天生支持css的样式隔离（vue组件中可给style标签指定scoped属性，自动添加唯一key为组件内的css指定作用域），而react需要通过css in js（[styled-components](https://link.zhihu.com/?target=https%3A//www.styled-components.com/)）或者css module的方式才能实现样式的隔离
* 兼容性：vue不兼容ie8
* 框架不同：vue本质是mvvm框架，react是前端组件化框架
* 相同点：他们都有状态管理工具，都是基于虚拟dom

### **受控组件和非受控组件的区别**

受控组件的状态由开发者维护，而非受控组件的的状态由组件自身维护

### **React context的作用，以及如何使用**

在一个典型的react应用中，数据都是通过props自上而下（从父组件到子组件）传递的，这种方式在某些嵌套很深的场景非常繁琐，需要逐层传递props。而context就是提供这样一种能力可以直接使用上层的props、组件之间共享数据的方式，而不用逐层传递

React.createContext创建context对象，该对象会从组件树中离自身最近的那个匹配的provider中读取到当前context的值

Context.Provider包裹订阅context变化的组件

### **React组件如何转发ref**

* class组件中通过creactRef的方式创建ref
* class组件中给dom中的ref赋值的方式
* react.forwardRef创建一个ref变量，传递给子组件ref

### **为什么出现了React hook，它解决了什么问题（这里可以参考react官方文档有说明），你使用过的react hook，为什么要保证hook的执行顺序**

1. class组件很难实现状态逻辑的复用
2. 复杂的组件变得难以理解：一个声明周期中掺杂各种不相关业务逻辑、需要同时在两个声明周期对事件监听做绑定和解绑
3. 难以理解的class：class语法糖使用有一定的复杂度

### **react hook缺点**

1. useEffect依赖链的问题：当useEffect依赖某个函数的不变行，而这个函数又依赖其他函数的不可变性，就形成了一条依赖链
2. 状态不同步：每个函数有独立的作用域。函数变量保存在运行时的作用域里面，当我们异步回调的时候经常会碰到异步回调的变量引用是之前的
3. hook不能写在条件判断的语句里面

为什么不可以呢？因为react把每次render中useState的顺序值0、1、2、3**当成了key**，方便后续render用key查找对应的state。因为保证了hooks的调用顺序，所以hooks内部可以使用链表来实现

### **React组件如何做性能优化 （这里包括几个方面，shouldComponentUpdate， PureComponent, React.memo, React.useCallback, React.useMemo, 以及key的使用）**

* purecomponent&shouldcomponentupdate&memo：父组件的state发生变化时，父组件重新渲染，也会导致子组件重新渲染，此时就可以使用shouldcomponentupdate关注子组件的state是否有变化来决定是否重新渲染，或者使用purecomponent来做一层props的浅比较（浅比较之前的前置比较：比如判断两个state长度是否相等、判断两个state里面包含的属性是否一样、从而优化浅比较的性能）。如果是函数式组件就使用memo
* 函数式组件每次执行，下面定义的变量和函数都会被重新创建，此时就可以用useCallback和useMemo来对变量做记忆化，仅当关注的变量发生变化时才重新创建这个变量或者函数

### **setState后发生了什么，它是同步的还是异步的**

react为了解决跨平台兼容问题，自己实现了一套事件机制代理了原生事件

setstate只有在合成事件以及生命周期中才是异步的（只要你进入了 react 的调度流程，那就是异步的。只要你没有进入 react 的调度流程，那就是同步的。什么东西不会进入 react 的调度流程？ setTimeout setInterval ，直接在 DOM 上绑定原生事件等。这些都不会走 React 的调度流程，你在这种情况下调用 setState ，那这次 setState 就是同步的。 否则就是异步的。）

我们调用 this.setState 的时候，对同一个值的更改，更新策略会创建一个队列，它会自动帮我们做一个 state 的合并，（而 hook 则不会）

### **为什么React diff的复杂度是O(n),难一点的会问diff算法**

<https://blog.csdn.net/qq_39207948/article/details/112639720>

如果整棵树的完整比较，复杂度高达O(N3),代表这如果有1000个节点所有需要的计算量就算10亿级这个范围，这个开销过大。为了降低复杂度，react采用了一些限制策略来降低复杂度

**diff限制策略**：

* 只对同层级的节点做diff比较，如果节点的变更已经跨越层级，则reac不会尝试复用他
* 因为节点类型不同，创建的虚拟dom结构不同，所以如果节点直接发生了类型的改变（比如从div变成了p），那就直接删除再重新创建
* 开发者可以通过key属性来暗示哪些元素可以复用

**不同类型diff如何实现：**

对于同级一个节点的diff

先判断节点的key是否相同，再判断节点类型是否相同，如果相同则可以复用

对于同级多个节点的diff

用新集合中的节点和老集合中的节点进行diff

* 如果diff结束后，新集合仍然有节点没有被diff，则新增该节点
* 如果diff结束后，老结合仍然有节点没有被diff，则删除该节点
* 如果diff结束后，有新节点以及老节没有被diff，则可能出现不可复用节点，则更新该节点
* 如果新老结合节点刚好diff完
  + 新旧节点位置相同，则无需再移动
  + 如果新旧节点位置不同，需要对老集合中的节点进行移动处理

首先设置一个游标lastIndex，首次指向新集合的第一个节点，从新集合第一个节点开始，寻找与旧集合匹配的节点，当匹配到的第一个旧集合节点索引小于新节点的索引(lastIndex)，则旧节点向后移，否则更新游标为旧节点的索引值，以此往复，直到游标变为最大索引并且旧集合遍历完。

尽量避免场景（集合中发生了最后一个节点移动到了第一个的情况），否则会对节点进行大量频繁的操作，影响react的渲染性能

### **说说React fiber机制**

fiber其实就是把一个线程再颗粒化的概念，也就是一个线程可以包含多个fiber

react的渲染页面的两个阶段

调度阶段：react生成虚拟dom，通过diff算法拿到需要更新的节点加入到更新队列，等待更新

渲染阶段：react拿到更新队列，将所有的变更一次性更新到dom上

由于调度阶段不可控，比如1000个组件需要更新，每个组件1ms，那就有将近1s的时间占用了主线程用来调度，这个阶段用户的任何操作都无法得到反馈，只有react的更新任务完成以后，主线程才会被释放。

也就是说一旦组件树过大，就会出现浏览器失去响应的情况，用户体验就不好

解决

所以提到了线程颗粒化的概念——fiber，fiber将大量的同步计算进行拆解和异步化，把一个耗时长的任务进行拆解，没一小片运行的时间很短，每隔一段时间就把主线程的操作权还给浏览器，让浏览器可以调控主线程。

阶段一，生成 Fiber 树，得出需要更新的节点信息。这一步是一个渐进的过程，可以被打断。

阶段二，将需要更新的节点一次过批量更新，这个过程不能被打断。

requestIdleCallback方法插入一个函数，这个函数将在浏览器空闲时期被调用。这使开发者能够在主事件循环上执行后台和低优先级工作，而不会影响延迟关键事件，如动画和输入响应。函数一般会按先进先调用的顺序执行，然而，如果回调函数指定了执行超时时间timeout，则有可能为了在超时前执行函数而打乱执行顺序。

window.requestAnimationFrame() 告诉浏览器——你希望执行一个动画，并且要求浏览器在下次重绘之前调用指定的回调函数更新动画。该方法需要传入一个回调函数作为参数，该回调函数会在浏览器下一次重绘之前执行.取消：cancelAnimationFram

### **如何处理React组件的异常**

针对渲染异常可以使用react错误边界进行处理（componentdidcatch，getderivedstatefromerror）

针对错误边界无法捕获的场景：事件处理、异步代码、服务端渲染、自身抛出的错误。使用try catch

### **React父组件如何调用子组件的方法（包括class组件和function组件）**

class组件直接通过ref拿到子组件的方法

function组件通过方法useimperativehandle实现，可以在使用ref时自定义暴露给组件的实例值



### **React组件事件如何绑定this**

类：constructor绑定this、使用箭头函数

组件函数：使用useref

### **实现React高阶组件的几种方式**

调用传入的组件（组合）

继承传入的组件（继承）

### **使用过React Router吗，说出你知道的几种router，各自区别和使用场景**

hashRouter：通过window.location.hash改变路由，通过hanshchange做监听

browserRouter：通过history的api，go,back,foward,pushstate、popstate、replacestate来改变路由，用popstate来做监听

staticRouter：静态路由，主要用于服务端渲染，服务端渲染不需要记录状态

hash特点：

hash 值是网页的标志位，HTTP 请求不包含锚部分，对后端无影响

因为 HTTP 请求不包含锚部分，所以 hash 值改变时，不触发网页重载

改变 hash 值会改变浏览器的历史记录

改变 hash 值会触发 window.onhashchange() 事件

### **做过React直出吗，说下直出的优缺点（同理CSR的优缺点），以及你是如何实现直出的**

**优缺点**

（1）客户端：

* 返回的首字节更快
* 降低服务端成本
* 白屏时间更长

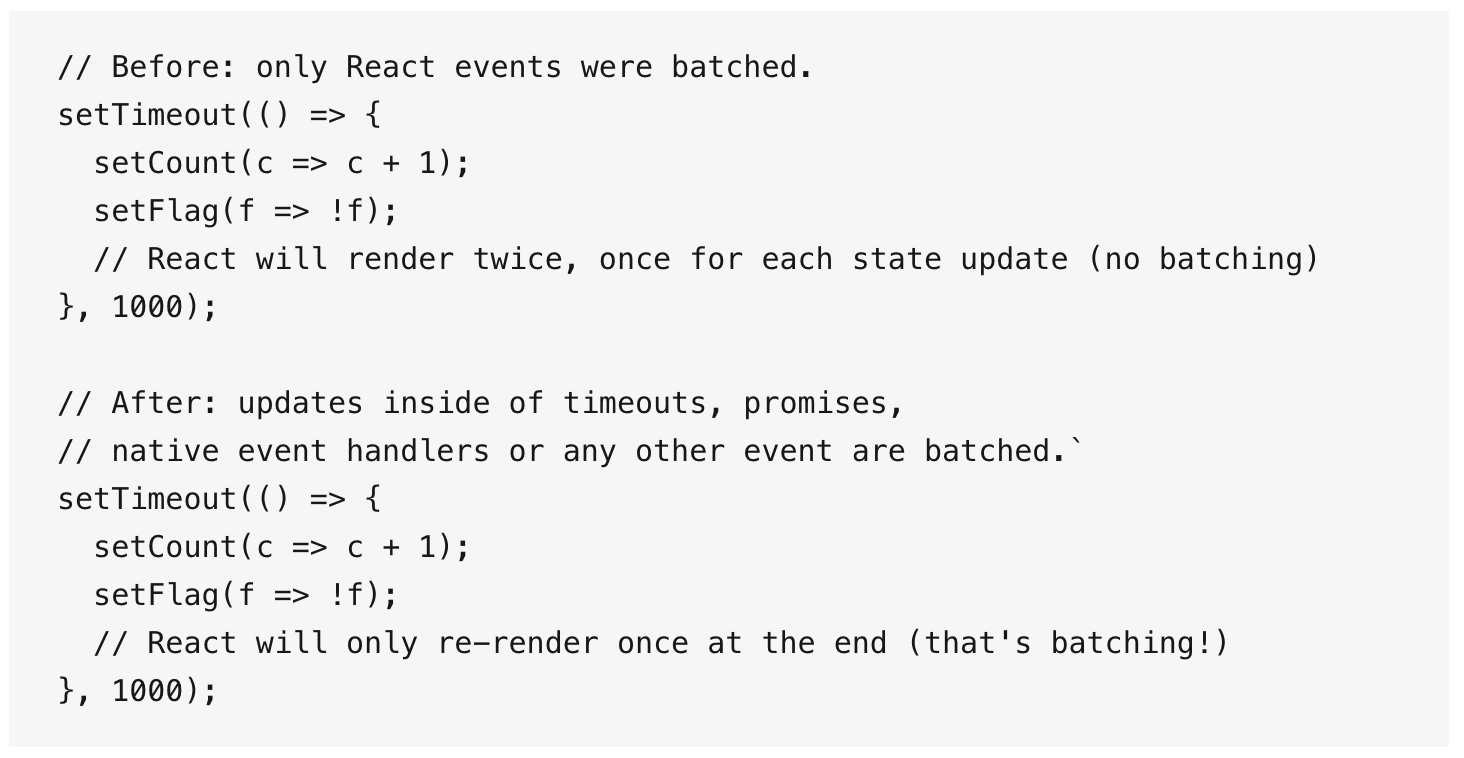
（2）服务端

* 白屏时间更短
* 利于搜索引擎检索，保证搜索结果可以排到更高位
* 更多工作从前端移交到服务端，增加服务端负载，原本只需要返回静态资源文件的服务器，现在增加获取数据的io，以及渲染html的cpu占用，如果出现流量激增，很有可能会down机，所以需要做好负载均衡
* 代码复杂度增加了，原本只用在客户端运行的代码，需要做兼容逻辑，保证能够在服务端渲染

### **说说React 18有哪些新特征**

（1）自动批处理

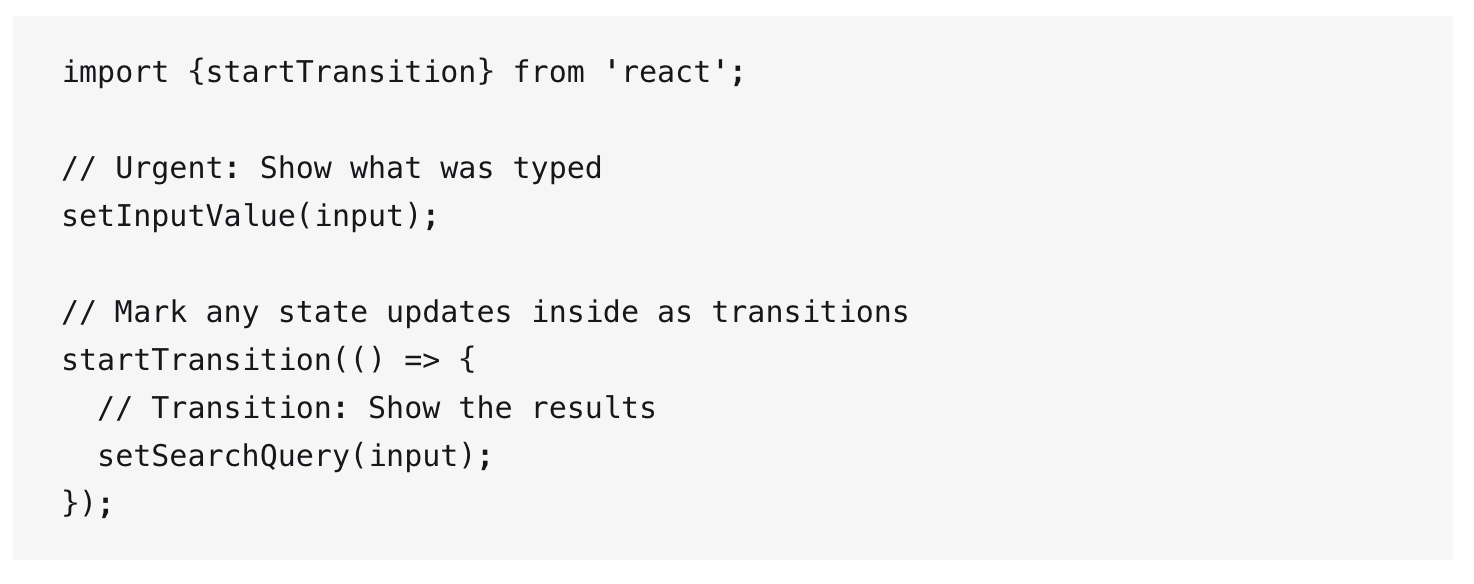
针对不在react调度流程中的行为（Promise、setTimeout、原生事件），这里对于数据的更新不会进行批处理，而新版支持了这个能力



（2）过渡

过渡是react一个新概念，用于区分紧急更新和非紧急更新。

用startTransition API 来通知 React 哪些更新是紧急的，哪些是过渡，包装在 startTransition 中的更新被视为非紧急更新，如果出现更紧急的更新，则会中断。过渡将选择并发渲染，这允许更新被中断。



（3）新的suspense特性

如果组件树的一部分暂时没有完全准备好，我们可以使用suspense以声明的方式指定它的加载状态

（4）新的客户端和服务端渲染api

* 客户端React DOM

这些新的 API 现在可以从 react-dom/client 导出：

* createRoot：创建要渲染或卸载的根的新方法。使用它来代替 ReactDOM.render。没有它，React 18 中的新功能就无法工作。
* hydraRoot：为服务器渲染的应用程序加水的新方法。将它与新的 React DOM 服务器 API 结合使用，而不是 ReactDOM.hydrate。没有它，React 18 中的新功能就无法工作。
* 服务器端React DOM

这些新的 API 现在可以从 react-dom/server 导出，并且完全支持在服务器上流式传输 Suspense：

* renderToPipeableStream：用于 Node 环境中的流式传输。
* renderToReadableStream：适用于现代边缘运行时环境，例如 Deno 和 Cloudflare worker。

现有的 renderToString 方法继续工作，但不鼓励。

### **useEffect和useLayoutEffect**

* useEffect 是异步执行的，而useLayoutEffect是同步执行的。
* useEffect 的执行时机是浏览器完成渲染之后，而 useLayoutEffect 的执行时机是浏览器把内容真正渲染到界面之前，和 componentDidMount 等价。

因为 useEffect 是渲染完之后异步执行的，所以会导致 hello world 先被渲染到了屏幕上，再变成 world hello，就会出现闪烁现象。而 useLayoutEffect 是渲染之前同步执行的，所以会等它执行完再渲染上去，就避免了闪烁现象。也就是说我们最好把操作 dom 的相关操作放到 useLayouteEffect 中去，避免导致闪烁。

因为 useLayoutEffect 是不会在服务端执行的，所以就有可能导致 ssr 渲染出来的内容和实际的首屏内容并不一致。

### **useEvent**

https://github.com/reactjs/rfcs/blob/useevent/text/0000-useevent.md

### **hippy**

Hippy 可以理解为一个精简版的浏览器，从底层做了大量工作，抹平了 iOS 和 Android 双端差异，提供了接近 Web 的开发体验，目前上层支持了 React 和 Vue 两套界面框架，前端开发人员可以通过它，将前端代码转换为终端的原生指令，进行原生终端 App 的开发。

同时，Hippy 从底层进行了大量优化，在启动速度、可复用列表组件、渲染效率、动画速度、网络通信等方面都提供了业内顶尖的性能表现。

### **有没有使用过React native，这里可以往hippy方向说(hippy&rn,hippy&flutter&原生)**

没有用过rn，用过hippy，是一款腾讯团队自研的类rn框架。

（1）相对rn，为什么使用hippy呢？

* 使用公司内部的自研框架是一种趋势，
* 并且遇到框架层面的问题时，可以找到开发的同学及时反馈并得到解决，
* hippy本身在rn的基础上做了一些性能的优化，所以性能上会更优一些。
* rn本身只支持react框架，而hippy同时支持react和vue

（2）为什么使用rn或者hippy，而不使用h5呢？

* 作为原生框架，它的体验更好更流畅
* 并且遇到框架层面的问题时，可以找到开发的同学及时反馈并得到解决，
* 由于h5生态更好，所以更新迭代会更快

（3）这种框架和flutter已经原生语言的对比

1. 原生语言需要针对ios和安卓双端分别做实现，成本更高，而rn和flutter只用做一次实现就可以在双端运行
2. flutter和原生语言的更新需要做版本发布，而rn可以只做热更新。流程上更简单
3. flutter是基于dart语言的开发，对于前端有额外的上手成本

### **jsbridge如何实现的**

<https://juejin.cn/post/6844904062102700045>

**注入 API 和 拦截 URL SCHEME。**

注入 API 方式的主要原理是，通过 WebView 提供的接口，向 JavaScript 的 Context（window）中注入对象或者方法，让 JavaScript 调用时，直接执行相应的 Native 代码逻辑，达到 JavaScript 调用 Native 的目的。

拦截 URL SCHEME 的主要流程是：Web 端通过某种方式（例如 iframe.src）发送 URL Scheme 请求，之后 Native 拦截到请求并根据 URL SCHEME（包括所带的参数）进行相关操作。

### **移动端chrome DevTools调试原理**

<https://zhaomenghuan.js.org/blog/chrome-devtools.html>

### **说说你对MVC、MVP、MVVM的理解**

三者都是框架模式，它们设计的目标都是为了解决Model和View的耦合问题。

MVC 模式出现较早主要应用在后端，如 Spring MVC、ASP.NET MVC 等，在前端领域的早期也有应用，如Backbone.js。它的优点是分层清晰，缺点是数据流混乱，灵活性带来的维护性问题。

MVP 模式在是 MVC 的进化形式，Presenter 作为中间层负责 MV 通信，解决了两者耦合问题，但 P 层过于臃肿会导致维护问题。

MVVM 模式在前端领域有广泛应用，它不仅解决 MV 耦合问题。viewmodel实现了一套更新策略，减少了dom操作代码，还同时解决了维护两者映射关系的大量繁杂代码和 DOM 操作代码，在提高开发效率、可读性同时还保持了优越的性能表现

### **react动态原生加载**

在 React 16.6 版本中，新增了 React.lazy 函数，配合 webpack 的 Code Splitting，只有当组件被加载，对应的资源才会导入 ，从而达到懒加载的效果。

React.lazy 需要配合 Suspense 组件一起使用，在 Suspense 组件中渲染 React.lazy 异步加载的组件。如果单独使用 React.lazy，React 会给出错误提示。

当所有的子组件对象都 resolve 后，Suspense再去替换所有子组件，这样也就避免了出现多个 loading 的体验问题

**lazy原理：**

webpack 通过、找出依赖对应的 chunk 信息，然后生成 script 标签来动态加载 chunk，每个 chunk 都有对应的状态：未加载、 加载中、已加载。chunk加载好以后更新状态为已加载

**suspend组件原理：**

父组件发现异步处理请求，抛出一个promise，suspend通过compnentdidcatch监听promise，当组件状态为pending是渲染fallback，当为resolve时渲染真正的组件

### **react为什么render的时候执行两次**

组件的一次更新流程，在视图真正刷新之前的部分都是可能被多次调用的，因而这些部分中不能出现副作用，开发环境下会刻意触发两次以使得开发者能注意到误用的副作用。

### **useRef**

useRef 在 react hook 中的作用,它像一个变量, 类似于 this , 它就像一个盒子, 你可以存放任何东西. createRef 每次渲染都会返回一个新的引用，而 useRef 每次都会返回相同的引用

### **react事件机制**

react上写的事件并没有绑定在对应的真实 DOM 上，而是通过事件代理的方式，将所有的事件都统一绑定在了 document 上。这样的方式不仅减少了内存消耗，还能在组件挂载销毁时统一订阅和移除事件。

另外冒泡到 document 上的事件也不是原生浏览器事件，而是 React 自己实现的合成事件（SyntheticEvent）。因此我们如果不想要事件冒泡的话，调用 event.stopPropagation 是无效的，而应该调用 event.preventDefault。

那么实现合成事件的目的是什么呢？

* 合成事件首先抹平了浏览器之间的兼容问题，另外这是一个跨浏览器原生事件包装器，赋予了跨浏览器开发的能力
* 对于原生浏览器事件来说，浏览器会给监听器创建一个事件对象。如果你有很多的事件监听，那么就需要分配很多的事件对象，造成高额的内存分配问题。但是对于合成事件来说，有一个事件池专门来管理它们的创建和销毁，当事件需要被使用时，就会从池子中复用对象，事件回调结束后，就会销毁事件对象上的属性，从而便于下次复用事件对象。

## redux

<https://juejin.cn/post/7114120958637506591>

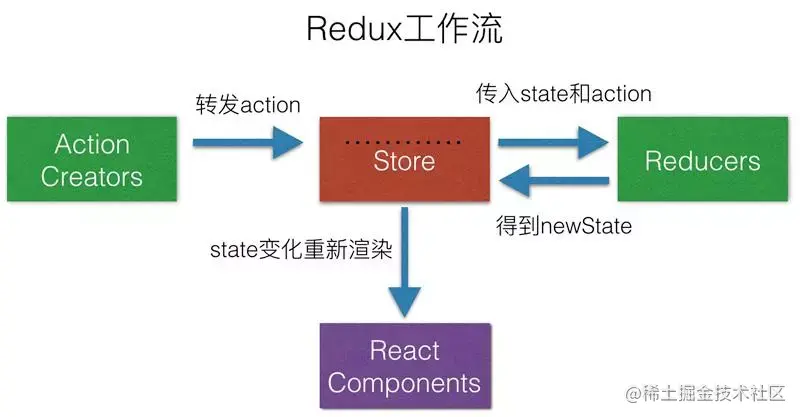
### **为什么需要Redux**

redux的主要优势之一是可以处理共享状态，如果说两个组件同时需要访问同一状态应该怎么处理呢？常见的操作是将状态管理放到父组件，然后再一层层往下传传递到子组件。但是这样做很不优雅，而且从父组件传递子组件的中间层组件对这个状态没有需求，只起到了传递的作用

而通过redux就可以在任何位置快速获取到该数据

### **Redux实现原理**

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的缺点**

redux的v6使用了react的reactcontext，将store中的state放在了context中，导致只要state发生变化，那其他依赖这个组件的context都会触发更新

解决的方式是使用reselect中间件来做处理。reselect类似于react 的usememo，对数据进行了记忆化，对整体store层级state进行缓存、组件级别state也进行缓存，之后进行一些浅比较的工作

### **用过的Redux 中间件（这里可能会问 Redux的异步如何处理， 一般都会回答Redux thunk、logger，接着还会问用过的其他处理异步的中间件，比如redux saga）**

redux thunk：需要找到一个“合法”的方法给辅助方法提供dispatch参数，并且帮助Redux区分出哪些是异步的action creator，好特殊处理他们。rtk已经内置了thunk

1. 一个中间件接收store作为参数，会返回一个函数
2. 返回的这个函数接收老的dispatch函数作为参数(也就是代码中的next)，会返回一个新的函数
3. 返回的新函数就是新的dispatch函数，这个函数里面可以拿到外面两层传进来的store和老dispatch函数

<https://segmentfault.com/a/1190000037437347>

### **它和mobx的区别，各自优缺点**

redux将数据保存在单一的store中，mobx将数据保存在分散的多个store中

redux使用plain object保存数据，需要手动处理变化后的操作；mobx适用observable保存数据，数据变化后自动处理响应的操作

redux使用不可变状态，这意味着状态是只读的，不能直接去修改它，而是应该返回一个新的状态，同时使用纯函数；mobx中的状态是可变的，可以直接对其进行修改

mobx相对来说比较简单，在其中有很多的抽象，mobx更多的使用面向对象的编程思维；redux会比较复杂，因为其中的函数式编程思想掌握起来不是那么容易，同时需要借助一系列的中间件来处理异步和副作用

mobx中有更多的抽象和封装，调试会比较困难，同时结果也难以预测；而redux提供能够进行时间回溯的开发工具，同时其纯函数以及更少的抽象，让调试变得更加的容易

### **react-redux中connect高阶组件的实现原理，它有什么问题，如何优化**

### **原理解析**

**首先connect之所以会成功，是因为Provider组件**：

* 在原应用组件上包裹一层，使原来整个应用成为Provider的子组件
* 接收Redux的store作为props，通过context对象传递给子孙组件上的connect

**那connect做了些什么呢？**

它真正连接 Redux 和 React，它包在我们的容器组件的外一层，它接收上面 Provider 提供的 store 里面的 state 和 dispatch，传给一个构造函数，返回一个对象，以属性形式传给我们的容器组件。

**关于它的源码**

* connect是一个高阶函数，首先传入mapStateToProps、mapDispatchToProps，然后返回一个生产Component的函数(wrapWithConnect)，然后再将真正的Component作为参数传入wrapWithConnect，这样就生产出一个经过包裹的Connect组件，该组件具有如下特点:
* 通过props.store获取祖先Component的store
* props包括stateProps、dispatchProps、parentProps,合并在一起得到nextState，作为props传给真正的Component
* componentDidMount时，添加事件this.store.subscribe(this.handleChange)，实现页面交互
* shouldComponentUpdate时判断是否有避免进行渲染，提升页面性能，并得到nextState
* componentWillUnmount时移除注册的事件this.handleChange

# 网络及操作系统

### **service worker**

Service Worker 是运行在浏览器背后的独立线程，一般可以用来实现缓存功能。使用 Service Worker的话，传输协议必须为 HTTPS。因为 Service Worker 中涉及到请求拦截，所以必须使用 HTTPS 协议来保障安全。

Service Worker 实现缓存功能一般分为三个步骤：首先需要先注册 Service Worker，然后监听到 install 事件以后就可以缓存需要的文件，那么在下次用户访问的时候就可以通过拦截请求的方式查询是否存在缓存，存在缓存的话就可以直接读取缓存文件，否则就去请求数据。

### **tcp和udp的区别**

* TCP 是面向连接的，udp 是无连接的即发送数据前不需要先建立链接。
* TCP 提供可靠的服务。也就是说，通过 TCP 连接传送的数据，无差错，不丢失， 不重复，且按序到达;UDP 尽最大努力交付，即不保证可靠交付。 并且因为 tcp 可靠， 面向连接，不会丢失数据因此适合大数据量的交换。
* TCP 是面向字节流，UDP 面向报文，并且网络出现拥塞不会使得发送速率降低 (因此会出现丢包，对实时的应用比如 IP 电话和视频会议等)。
* TCP只能是1对1的，UDP支持1对1,1对多。
* TCP 的首部较大为 20 字节，而 UDP 只有 8 字节。

### 三次握手四次挥手

* SYN: 同步序号，表示请求建立连接，设置序列号字段初始值，SYN=1
* ACK: ACK=1 确认号有效，ACK=0，确认号无效
* ack: 确认号，序列号seq是每个字节的第一位，而确认号希望接受到下一个字节的编号 ack=上一次序列号+1
* seq: 序列号，字节的第一位编号，随机生成
* FIN：希望断开连接

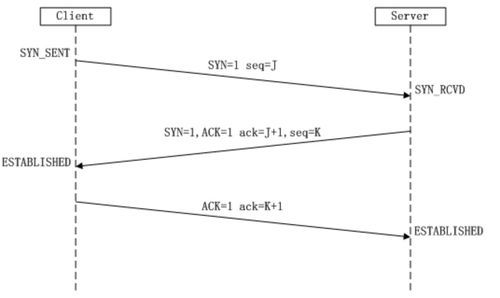
#### 三次握手

第一次握手：Client将标志位SYN置为1，随机产生一个值seq=J，并将该数据包发送给Server，Client进入SYN\_SENT状态，等待Server确认。

SYN：同步序列编号(Synchronize Sequence Numbers)

第二次握手：Server收到数据包后由标志位SYN=1知道Client请求建立连接，Server将标志位SYN和ACK都置为1，ack=J+1，随机产生一个值seq=K，并将该数据包发送给Client以确认连接请求，Server进入SYN\_RCVD状态。

第三次握手：Client收到确认后，检查ack是否为J+1，ACK是否为1，如果正确则将标志位ACK置为1，ack=K+1，并将该数据包发送给Server，Server检查ack是否为K+1，ACK是否为1，如果正确则连接建立成功，Client和Server进入ESTABLISHED状态，完成三次握手，随后Client与Server之间可以开始传输数据了。



#### 为什么需要三次握手

client发出的第一个连接请求报文段并没有丢失，而是在某个网络结点长时间的滞留了，以致延误到连接释放以后的某个时间才到达server。本来这是一个早已失效的报文段。但server收到此失效的连接请求报文段后，就误认为是client再次发出的一个新的连接请求。于是就向client发出确认报文段，同意建立连接。假设不采用三次握手，那么只要server发出确认，新的连接就建立了。由于现在client并没有发出建立连接的请求，因此不会理睬server的确认，也不会向server发送数据。但server却以为新的运输连接已经建立，并一直等待client发来数据。这样，server的很多资源就白白浪费掉了。采用三次握手的办法可以防止上述现象发生

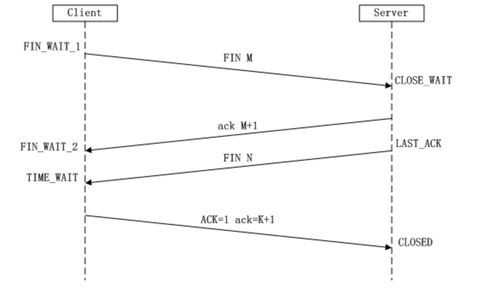
#### 四次挥手

1.第一次挥手：Client发送一个FIN，用来关闭Client到Server的数据传送，Client进入FIN\_WAIT\_1状态。

2.第二次挥手：Server收到FIN后，发送一个ACK给Client，确认序号为收到序号+1（与SYN相同，一个FIN占用一个序号），Server进入CLOSE\_WAIT状态。

3.第三次挥手：Server发送一个FIN，用来关闭Server到Client的数据传送，Server进入LAST\_ACK状态。

4.第四次挥手：Client收到FIN后，Client进入TIME\_WAIT状态，接着发送一个ACK给Server，确认序号为收到序号+1， Server进入CLOSED状态，完成四次挥手。



#### 为什么建立是3次握手，而关闭是4次挥手呢

因为建立连接的时候，客户端接受的是syn + ack包。而关闭的时候，服务端接受fin后，客户端仅仅是不再发送数据，但是还是可以接收数据的。服务端此时可以选择立刻关闭连接，或者再发送一些数据之后，再发送fin包来关闭连接。因此fin与ack包一般都会分开发送。

#### TCP三次握手中可以传输数据吗

第一次第二次不可以，因为很容易被人攻击，而第三次握手已经进入establish状态了，已经确认双方的收发能力，可以传输

### TCP是怎么实现可靠传输的

1. 校验和: 传输的过程将数据段都当成16位的整数，然后将这些整数相加，且不能丢弃进位，补在最后，最后取反得到校验和
2. 序列号: TCP在传输的过程中对每个字节都进行了编号
3. 确认应答: TCP传输过程中，每次接收方收到数据后，都会对传输方进行确认应答,也就是发出ACK包
4. 超时重传:: 超时重传可以理解为发送方在发送完数据后等待一段时间，时间到达但没有接收到ACK报文，那么对刚才的数据进行重新发送
5. 连接管理: 三次握手与四次挥手的过程，是为了保证可靠地连接，是保证可靠性的前提
6. 流量控制: 接受方在处理数据时，如果发送方发送数据的速度太快，就会导致接收端的结束缓冲区很快就会占满，如果继续发送数据，那么接下来的数据都会丢包，进而进行丢包的一系列反应。这时滑动窗口就起到了作用，提前告知发送方自己的窗口大小，从而控制对方的发送速度
7. 拥塞控制: 发送端在发送大量的数据时，可能会造成大面积的拥堵。拥堵的加剧就会导致大量的丢包，然后就会有大量的超时重传，严重影响传输。所以TCP加入慢启动的概念，首先在发送数据时，先发送少量的数据进行探路，摸清楚当前的网络状况，然后决定以多大速度进行传输。然后就引入了拥塞窗口的概念，发送数据之前，首先将拥塞窗口的大小和接收端反馈窗口对比，取最小的值作为实际发送的窗口大小，另外拥塞窗口还可以设置阈值，慢启动会将阈值减为原来的一半。

### http协议的特点

* 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。
* 可靠传输 HTTP 基于 TCP/IP，因此把这一特性继承了下来。这属于 TCP 的特性，不具体介绍了。
* 请求-应答 也就是一发一收、有来有回， 当然这个请求方和应答方不单单指客户端和服务器之间，如果某台服务器作为代理来连接后端的服务端，那么这台服务器也会扮演请求方的角色。
* 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

### http1、http2、http3

**http1.0**

* 无法复用链接，每次发送请求都需要进行一次tcp链接，使得网络利用率变低
* 并且还存在队头阻塞的问题，因为http1.0规定，下一个请求必须在上一个请求响应到达之后才能发送，如果前一个请求一直不到大，那么下一个请求就一直无法发送，导致阻塞
* 也不支持断点续传，每次都会传送全部的页面和数据

**http1.1**

* 开始支持长连接，那对于同一个host，只要设置了connection:keep-alive，就可以保证链接不断，提高了网络利用率在一个TCP连接上可以传送多个HTTP请求和响应，减少了建立多个连接的消耗和延迟
* 支持断点续传，通过请求头的range来实现
* 新增了24个错误状态响应码，如409（conflict）表示请求的资源与资源当前的状态发生冲突，401（gone）表示服务器上的资源被永久的删除。
* 可以使用管道传输，多个请求可以同时发送，但是还是顺序响应，如果前面的请求没有响应，后面的请求就有可能阻塞，又加【队头阻塞】

HTTP1.0版本存在一些问题，比如说连接无法复用的问题，这会导致每发送一次请求都需要进行三次握手的过程，重新建立连接，效率太低。还有就是阻塞问题，http1.0是下一个请求的发送必须要等到上一个请求返回后才会进行，如果上一个请求没有返回，那么后面的请求就会全部阻塞。最后就是安全问题，http1.0所传输的内容都是明文的，无法保证数据的安全性。

**http2.0**

* 采用了二进制分帧的方式来传输数据HTTP1.X的解析是基于文本，基于文本协议的格式解析存在天然缺陷，文本的表现形式有多样性，要做到健壮要考虑的场景必然很多。

HTTP2.0采用的是二进制格式，只认0和1的组合。实现方便且健壮

* 实现了多路复用，数据传输基于流的机制，每个数据流的消息分成一个或多个帧，他们可以是乱序发送的，之后再根据帧头部的标识符重新组装顺序。针对同一个域名只建立一个tcp链接。HTTP/1.x只能在一个TCP连接上发送一个HTTP请求，浏览器对同源HTTP/1.x连接的并发个数有限制，典型值是6
* 对请求头进行了压缩
* 服务端主动push

HTTP2.0版本比较专注于性能，它采用二进制格式传输数据，http2.0也采用了多路复用的技术，它可以只通过一个 TCP 连接传输所有的请求数据。还有就是采用了头部压缩技术，这也成功解决了http1.0的header 里携带的内容过大的问题，在一定程度上减轻了传输的成本。不过它也存在一定的问题，如果在传输的过程中存在丢包的情况的话，那么整个tcp就得重新传输，后面资源就会被阻塞。

**http3.0**

**核心思想是将tcp协议在内核实现的诸如可靠传输、流量控制、拥塞控制等功能转移到用户态来实现。同时在加密传输方向的尝试也推动了tls1.3的发展（tls1.3首次连接只需要1rtt，而1.2需要2rtt；而重复链接tls1.2需要1rtt，1.3确实0rtt）**

它最大的不同是放弃了tcp协议而是改用了 QUIC协议，此协议基于传输层UDP协议。UDP协议无需三次握手四次挥手，所以传输速率更高。并且它改善了多路复用产生的问题，如果出现丢包的情况，不需要整个重新发送，只需要重发丢掉的包就可以。

QUIC的实现，依然要面对消息的可靠性、滑动窗口、拥塞控制等场景，QUIC能够实现TCP协议的所有功能性需求，并集成了TLS，功能上赶超了TCP

rtt：数据包一来一回的时间消耗

### dns工作原理

DNS 协议提供的是一种主机名到 IP 地址的转换服务，就是我们常说的域名系统。是应用层协议，通常该协议运行在UDP协议之上，使用的是53端口号。

DNS在本地DNS服务器是如何查询的

* 输入域名，操作系统先检查host文件有没有，没有就会查询本地 DNS 服务器（一般是网络接入服务商提供 eg:电信，移动）
* 本地 DNS 服务器会首先查询它的缓存记录，若有，则直接返回结果，若没有，本地 DNS 服务器还要向 DNS 根服务器进行查询；
* DNS 根服务器返回映射关系，或者告诉本地 DNS 服务器，顶级域名服务器地址
* 本地服务器继续向顶级域名服务器发出请求，顶级域名服务器返回映射关系或者返回下一级域名服务器的ip（权限域名服务器）
* 本地 DNS 从域名服务器，收到域名与 IP 地址对应关系
* 本地 DNS 服务器将 IP 地址返回电脑，且保存副本到缓存已备下次查询。

### DNS为什么使用udp作为传输层协议

**「DNS 使用 UDP 协议作为传输层协议的主要原因是为了避免使用 TCP 协议时造成的连接时延。」**

* 为了得到一个域名的 IP 地址，往往会向多个域名服务器查询，如果使用 TCP 协议，那么每次请求都会存在连接时延，这样使 DNS 服务变得很慢。
* 大多数的地址查询请求，都是浏览器请求页面时发出的，这样会造成网页的等待时间过长。

### DNS污染及预防

污染：篡改hosts文件映射关系或者权限域名服务的缓存

解决：

1、dns的ttl值设置的尽量小，减小生存周期

2、禁止解析客户端和服务端的hosts文件

3、配置dns over https（因为域名解析过程没有任何加密，所以容易被攻击。使用加密的https协议进行dns解析请求，避免请求被窃听或者修改。但是本事https协议初始化比较耗时）

### CDN解决的问题

**CDN解决了什么问题?**

内容分发网络（CDN）是一组分布在多个不同地理位置的 Web 服务器。我们都知道，当服务器离用户越远时，延迟越高。CDN 就是为了解决这一问题，在多个位置部署服务器，让用户离服务器更近，从而缩短请求时间。

**CND原理**

当用户访问一个网站时，如果没有 CDN，过程是这样的：

* 浏览器要将域名解析为 IP 地址，所以需要向本地 DNS 发出请求。
* 本地 DNS 依次向根服务器、顶级域名服务器、权限服务器发出请求，得到网站服务器的 IP 地址。
* 本地 DNS 将 IP 地址发回给浏览器，浏览器向网站服务器 IP 地址发出请求并得到资源。

如果有 CND 加持

* 浏览器要将域名解析为 IP 地址，所以需要向本地 DNS 发出请求。
* 本地 DNS 依次向根服务器、顶级域名服务器、权限服务器发出请求，得到**全局负载均衡系统**（GSLB的 IP 地址。
* 本地 DNS 再向 GSLB 发出请求，GSLB 的主要功能是根据本地 DNS 的 IP 地址判断用户的位置，筛选出距离用户较近的本地负载均衡系统（SLB），并将该 SLB 的 IP 地址作为结果返回给本地 DNS。
* 本地 DNS 将 SLB 的 IP 地址发回给浏览器，浏览器向 SLB 发出请求。
* SLB 根据浏览器请求的资源和地址，选出最优的缓存服务器发回给浏览器。
* 浏览器再根据 SLB 发回的地址重定向到缓存服务器。
* 如果缓存服务器有浏览器需要的资源，就将资源发回给浏览器。如果没有，就向源服务器请求资源，再发给浏览器并缓存在本地。

### TCP的keepAlive和HTTP的keep-alive的区别问题

http keep-alive与tcp keep-alive，不是同一回事，意图不一样。http keep-alive是为了让tcp活得更久一点，以便在同一个连接上传送多个http，提高socket的效率。而tcp keep-alive是TCP的一种检测TCP连接状况的保鲜机制

### 浏览器缓存

缓存的优点：

* 减少了不必要的数据传输，节省带宽
* 减少服务器的负担，提升网站性能
* 加快了客户端加载网页的速度
* 用户体验友好

缺点：

* 资源如果有更改但是客户端不及时更新会造成用户获取信息滞后，如果老版本有bug的话，情况会更加糟糕。

**浏览器缓存分为：强缓存和协商缓存**

**强缓存**

当浏览器去请求某个文件的时候，服务端就在respone header里面对该文件做了缓存配置。缓存的时间、缓存类型都由服务端控制，具体表现为：

respone header 的cache-control，常见的设置是max-age public private no-cache no-store等

强缓存总结

1. cache-control: max-age=xxxx，public

客户端和代理服务器都可以缓存该资源；

客户端在xxx秒的有效期内，如果有请求该资源的需求的话就直接读取缓存,statu code:200 ，如果用户做了刷新操作，就向服务器发起http请求

1. cache-control: max-age=xxxx，private

只让客户端可以缓存该资源；代理服务器不缓存

客户端在xxx秒内直接读取缓存,statu code:200

1. cache-control: max-age=xxxx，immutable

客户端在xxx秒的有效期内，如果有请求该资源的需求的话就直接读取缓存,statu code:200 ，即使用户做了刷新操作，也不向服务器发起http请求

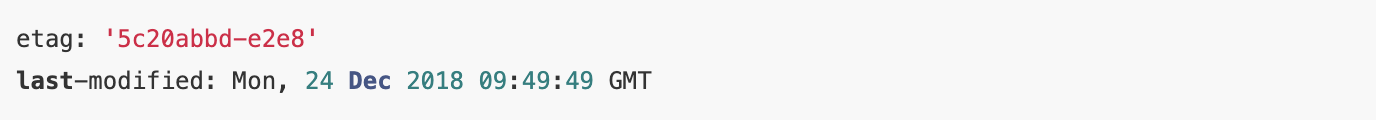
1. cache-control: no-cache

跳过设置强缓存，但是不妨碍设置协商缓存；一般如果你做了强缓存，只有在强缓存失效了才走协商缓存的，设置了no-cache就不会走强缓存了，每次请求都回询问服务端。

1. cache-control: no-store

不缓存，这个会让客户端、服务器都不缓存，也就没有所谓的强缓存、协商缓存了。

**弱缓存**

****

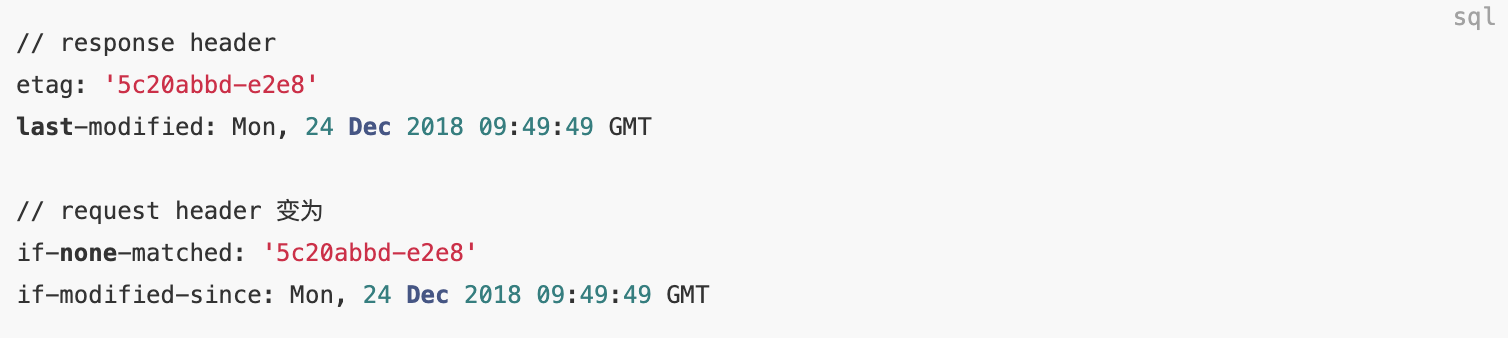
etag：每个文件有一个，改动文件了就变了，就是个文件hash，每个文件唯一，就像用webpack打包的时候，每个资源都会有这个东西，如： app.js打包后变为 app.c20abbde.js，加个唯一hash，也是为了解决缓存问题。

last-modified：文件的修改时间，精确到秒

每次请求返回来 response header 中的 etag和 last-modified，在下次请求时在 request header 就把这两个带上，服务端把你带过来的标识进行对比，然后判断资源是否更改了，如果更改就直接返回新的资源，和更新对应的response header的标识etag、last-modified。如果资源没有变，那就不变etag、last-modified，这时候对客户端来说，每次请求都是要进行协商缓存了

* 发请求-->看资源是否过期-->过期-->请求服务器-->服务器对比资源是否真的过期-->没过期-->返回304状态码-->客户端用缓存的老资源。
* 发请求-->看资源是否过期-->过期-->请求服务器-->服务器对比资源是否真的过期-->过期-->返回200状态码-->客户端如第一次接收该资源一样，记下它的cache-control中的max-age、etag、last-modified等。

补充一点，response header中的etag、last-modified在客户端重新向服务端发起请求时，会在request header中换个key名：



**为何既有last-modified又有Etag**

考虑以下情况：

1. 一些文件也许会周期性的更改,但是他的内容并不改变(仅仅改变的修改时间),这个时候,我们并不希望客户端认为这个文件被修改了,而重新 get
2. 某些文件修改非常频繁,比如在秒以下的时间内进行修改(比方说 1s 内修改了 N 次),If-Modified-Since能检查到的粒度时 s 级的,这种修改无法判断(或者说 UNIX 记录 MTIME只能精确到秒)
3. 某些服务器不能精确得到的文件的最后修改时间

### 五层网络协议

* 物理层：把通信主机连接起来的物理手段，比如双绞线、同轴电缆、光纤等。作用是来传输1010…的的电信号，数据单位是比特。
* 数据链路层：简称链路层。主机间传输数据，总是一段一段在链路上传送的。链路层负责使用链路层协议将网络层传输下来的 IP 数据组装成数据帧，并在链路上传输。
* 网络层：网络层使用 IP 协议为各主机之间提供通信服务。它会把运输层提供的报文分段/用户数据报包装成分组（包/IP数据报）进行传输。网络层还有一个重要的功能：选择合适的路由，以便源主机的数据能通过路由传输到目的主机。我们常说的路由器就是工作在这一层。
* 运输层：运输层负责为两个主机进程之间的通信提供数据传输服务，应用进程利用该服务来传输应用层报文。传输层主要使用的是 TCP 协议（传输单位是报文段）、UDP 协议（传输单位是用户数据报）。
* 应用层：直接为用户的应用进程（运行的程序）提供服务
* OSI分层 （7层）：物理层、数据链路层、网络层、传输层、会话层、表示层、应用层。
* TCP/IP分层（4层）：网络接口层、 网际层、运输层、 应用层。
* 五层协议 （5层）：物理层、数据链路层、网络层、运输层、 应用层。

### option作用

* OPTIONS 请求与 HEAD 类似，一般也是用于客户端查看服务器的性能。
* 这个方法会请求服务器返回该资源所支持的所有 HTTP 请求方法，该方法会用'\*'来代替资源名称，向服务器发送 OPTIONS 请求，可以测试服务器功能是否正常。
* JS 的 XMLHttpRequest对象进行 CORS 跨域资源共享时，对于复杂请求，就是使用 OPTIONS 方法发送嗅探请求，以判断是否有对指定资源的访问权限。

### 队头阻塞

对于每一个HTTP请求而言，这些任务是会被放入一个任务队列中串行执行的，一旦队首任务请求太慢时，就会阻塞后面的请求处理，这就是HTTP队头阻塞问题。

**解决办法:**

* 并发连接: 对每个域名分配长连接，那么可以理解成增加了任务队列，也就是说不会导致一个任务阻塞了该任务队列的其他任务
* 域名分片: 多个域名指向同一个服务器

### http和https区别

* HTTP 是明文传输协议，HTTPS 协议是由 SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，比 HTTP 协议安全。
* HTTPS 比 HTTP 更加安全，对搜索引擎更友好，利于 SEO,谷歌、百度优先索引HTTPS网页。
* HTTPS标准端口443，HTTP标准端口80。
* HTTPS需要用到SSL证书，而HTTP不用。

### https加密过程

* 服务端将自己的公钥发送给证书机构
* 证书颁发机构用自己的私钥加密服务端的公钥，然后用服务端网址等信息生成证书签名，同时也用私钥加密。
* 制作好以后将证书发给服务端，服务端将证书发给客户端
* 客户端确定是哪个证书机构以后，用证书机构的公钥解密出证书签名，然后用同样的规则生成一个证书签名，如果两者相匹配，则说明证书有效
* 接着用公钥解密出服务端的公钥，客户端再自己生成一个对称秘钥，用该公钥加密，然后发给服务端
* 服务端收到后用自己的私钥解密，拿到秘钥，之后就用该秘钥通信

### 进程和线程的区别

* 线程是程序执行的最小单位，而进程是操作系统分配资源的最小单位。
* 一个进程由一个或多个线程组成，线程是一个进程中代码的不同执行路线。
* 进程之间相互独立，但同一进程下的各个线程之间共享程序的内存空间。
* 调度和切换：线程上下文切换比进程上下文切换要快得多

**Chrome为什么从单进程转成多进程架构**

首先早期的浏览器都是单进程的，浏览器的主要组成部分有，浏览器引擎，呈现引擎，用户界面后端，网络，jascript解析器等模块，多个模块放在一个进程下可能会导致以下问题

* **不稳定** 根据操作系统知识，一个进程下的多个线程只要有一个线程发生错误，整个进程就会崩溃。可想而知，如果浏览器是单进程的话，随便哪个插件还是哪个模块崩溃，会导致整个浏览器都无法正常运行，这样是很危险的，除此之外渲染引擎也是如此
* **响应慢**单进程浏览器，所有页面都会在一个线程渲染，如果碰到某个页面存在无限渲染脚本，就会导致其他页面无法显示，变卡顿

### 进程通信方式

* 管道( pipe )：管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。
* 有名管道 (named pipe) ： 有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。
* 信号量( semophore ) ： 信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。
* 消息队列( message queue ) ： 消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。
* 信号 ( sinal ) ： 信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。
* 共享内存( shared memory ) ：共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号两，配合使用，来实现进程间的同步和通信。
* 套接字(socket) ： socket也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同及其间进程通信。

### 什么是死锁

死锁是指**两个或两个以上的进程在执行过程中**，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。

**解决办法**:

* **加锁顺序**:确保所有的线程都是按照相同的顺序获得锁
* **加锁时限**:在尝试获取锁的过程中若超过了这个时限该线程则放弃对该锁请求
* **死锁检测**:检测出来就释放所有的锁，回退，并且等待一段随机的时间后重试

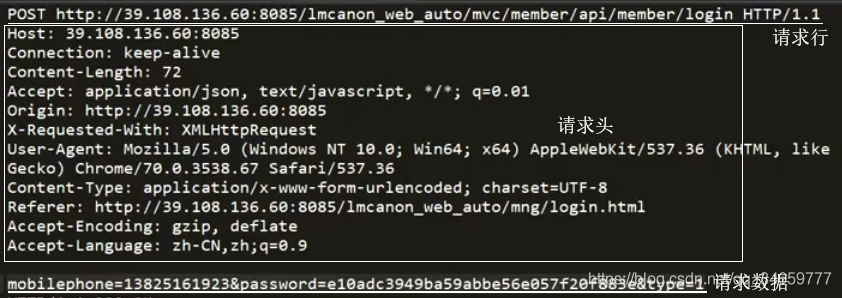
### http常见状态码及请求头介绍

<https://blog.csdn.net/qq_34659777/article/details/82753594>

* 1. **请求报文**

一个http请求报文由请求行、请求头、空行、请求数据组成



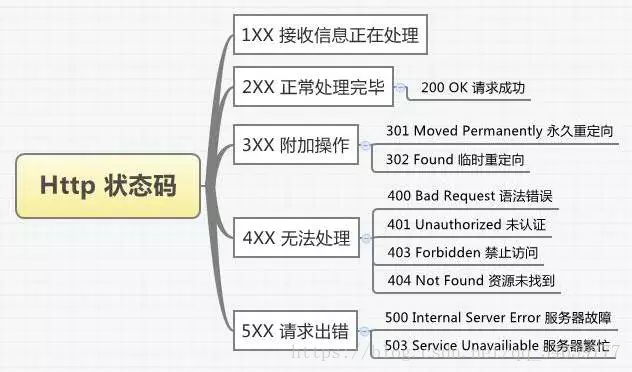


* 1. **响应报文**

一个http请求报文由状态行、响应头、空行、响应数据组成

* 1. **状态码**

****

****

* 1. **请求头信息**

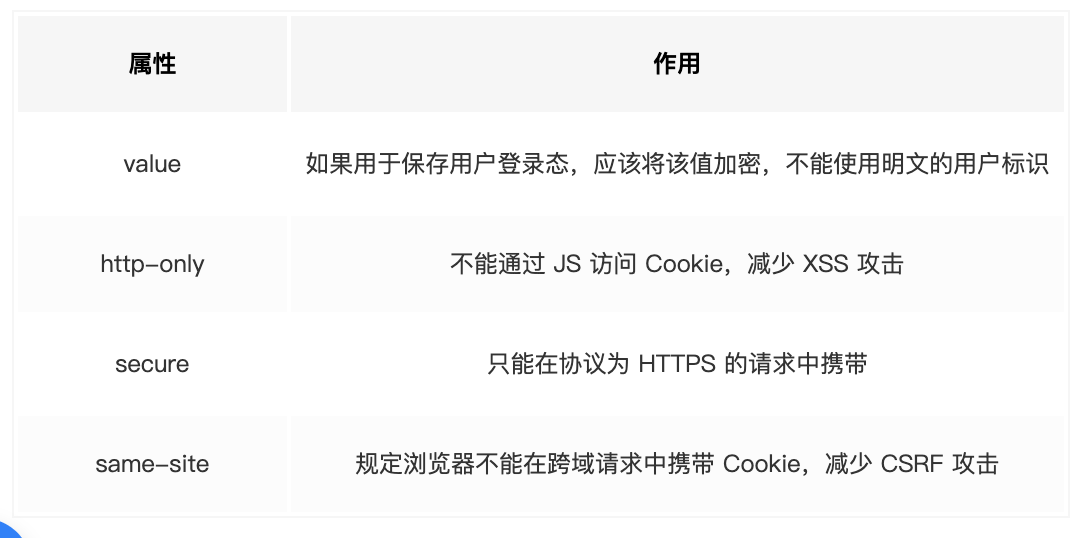
client头域：client头域的作用，客户端和服务器端要进行通信，那么客户端就得向服务器申明自己的浏览器版本、使用的操作系统（User-Agent）、交流的语言（Accept-Language）、交流的编码方式（accept-Encoding），以及客户端接受的内容媒体类型（accept）等

Cache头域：If-Modified-Since、If-None-Match、Cache-Control

transport头域：connect(keep-alive、close)、host(用于指定被请求资源的主机名和端口号)

cookie头域（写法：Set-Cookie：cookieName=cookieValue）

常见cookie属性：HttpOnly(设置以后客户端无法通过脚本方式读写cookie，防xss)、Expires(过期时间)、Max-Age(有效时间长度，优先级高于Expires、单位为秒)、path(决定cookie在哪些域名下携带)



### 断点续传的实现

https://juejin.cn/post/6844904046436843527

# 后台相关

## nodejs

### eventBus实现

### nodejs的优缺点

**优点：**

* 适合高并发的场景：多线程方式会出现阻塞的场景，nodejs异步机制，在事件驱动的高并发场景下性能更好
* 适合io密集型应用：进行非阻塞的io处理
* 本身轻量高效

**缺点：**

* 不适合cpu密集型应用：cpu密集型操作给nodejs带来的挑战是，由于js是单线程的，如果有长时间运行的计算，将导致cpu时间片无法释放

解决：分解大型运算任务为多个小任务，让运算能够适时释放，不阻塞io调用的发起

* 只支持单核cpu，不能充分利用多核cpu服务器，一旦这个进程崩掉了，那整个服务也就崩掉了

解决：

* 做nginx反向代理、负载均衡
* 开多个进程监听同一个端口

### 能否说下nodejs的事件循环，它与浏览器的事件循环有什么不一样的地方

相对浏览器环境，多了setImmediate和proccess.nextTick

1. timers： 执行setTimeout和setInterval到了时间的回调
2. IO callback:io的回调
3. idle，prepare：nodejs内部使用
4. poll：轮询阶段
5. check：执行setImmediate的阶段
6. close callback：执行close事件的 callback，例如socket.on('close',fn)等

setImmediate：设计是**当前轮询阶段完成后就立即执行**，如果放在同一个I/O循环内使用，setImmediate总是被优先调用，他的优势是如果在I/O周期内被调度的话，他会在所有定时器之前执行。他执行的方式是当前任务队列的尾部

process.nextTick：触发的是在**当前执行栈的尾部**，下一次事件循环之前，总是在**异步之前**。

为什么要使用process.nextTick()，因为可以允许用户处理错误，或者是在事件循环继续之前重试请求

### 流是什么，它有什么特点

流是一种在node.js中处理流式数据的抽象接口，流就是为我们提供一套处理数据的工具

### nodejs和其他多线程语言（php， java等）处理请求的方式有什么不同，各自有什么优缺点

nodejs更加适合io密集型的操作、但是不适合处理cpu密集型的操作

nodejs是解释型语言、java是编译型语言

nodejs学习曲线更简单

### 如何充分利用多个CPU

使用master-worker主从模式，node提供了child\_process模块，并且也提供了fork()方法来实现子进程的创建，主进程不负责具体的业务处理，而是负责调度和管理工作进程，工作进程负责具体的业务处理，所以，工作进程的稳定性是开发人员需要关注的。通过fork()复制的进程都是一个独立的进程，这个进程中有着独立而全新的V8实例。

* 使用 child\_process 模块并在一个子进程中运行 CPU 密集型
* 使用 cluster 模块，在多个进程中运行多个 CPU 密集型操作（cluster底层是基于process，但是做了一些负载均衡的操作，协调各个进程之间的负载。就是分布式架构中用于并行处理业务的逻辑，就有更好的伸缩性和稳定性）
* 使用worker线程，通过运行应用使用多个相互隔离的 workers 来实现并发，每一个 worker 将拥有其自己的 V8 实例及事件循环（Event Loop）。但和  child\_process 不同的是，workers 不共享内存。

### 介绍一下require的模块加载机制

require都是模块化编程的诞生物，是 AMD规范引入方式，提供了服务器/浏览器的模块加载方案。只能在运行时确定模块的[依赖关系](https://so.csdn.net/so/search?q=%E4%BE%9D%E8%B5%96%E5%85%B3%E7%B3%BB&spm=1001.2101.3001.7020)及输入/输出的变量，无法进行静态优化。require加载模块可分为下面三种情况

1. **加载核心模块**

核心模块默认已经被封装编译到二进制文件中了，直接使用加载就行；

例如require('fs')、require('http')

1. **加载本地模块（自己写的模块）**

必须以路径的形式加载，否则node会把他当成是核心模块或者第三方模块加载，文件后缀.js可以忽略。

例如：require('./a.js')、require('../b.js')

1. **加载第三方模块（通过npm下载安装的模块）**

为什么第三方模块不会和核心模块加载冲突？

因为不可能有第三方模块名会和核心模块名一样（规定）

* + 1. 找node\_module文件下对应模块名的文件夹
    2. 找package.json文件
    3. 找main属性确定入口模块进行加载，如果不存在mian，node就会去找目录下的index.js文件，如果还找不到就会去上一级目录找，知道找到根目录为止，如果还找不到就报错

### commonjs和esm的区别

1. 介esm输出的值的引用，commonjs输出的值的拷贝
2. commonjs是运行时加载，ems有一个静态分析阶段，之后编译执行
3. commonjs是同步加载，ems是异步加载：因为commonjs需要记载的模块都在本地，所以采用同步加载的方式也不会出问题，但是esm用于浏览器端时，如果采用 同步的方式可能导致阻塞，导致页面响应不及时

### 绍一下nodejs的垃圾回收机制

扫描法、标记清除、标记整理。堆内存、堆外内存。

v8的内存相对较小的原因是：

1. v8最初为浏览器而设计的，不太可能遇到需要太大内存的场景

2）v8垃圾回收机制的限制，每次垃圾回收都会导致js线程执行暂停，如果太大的话，垃圾回收很慢，应用的响应能力和性能会下降

垃圾回收机制

v8的整体内存为新生代和老生代所用的内存空间总和，其中新生代中的对象为存活时间较短的对象

1. **scavenge扫描算法**

新生代主要使用扫描算法，他将堆内存一分为二，两部分空间只有一处在使用中，另一处处于闲置的状态，处于使用中的状态为**FROM空间**，限制状态的为**TO空间**，分配对象时，先从FROM空间中进行分配。

垃圾回收时，先从FROM空间中查找存活的对象，将其复制到TO空间中，非存活的对象就会被释放掉

扫描算法存在的问题：对象多时复制对象的效率低、浪费了一半的空间。所以扫描算法是典型的用空间换取时间的算法

**晋级**

需要将存活时间较长的对象移到老生代中，完成对象的晋级；对象晋级的条件有两个：

* 是否经历过扫描回收
* TO空间的内存占用超过限制（如果转移时发现to空间的内存占比只剩25%，即直接晋升老生代）

1. **标记清除&标记整理**

老生代主要使用这两种算法

**标记清除**：遍历堆中所有的对象，标记活着的对象，清除没有被标记的对象。该方式会造成内存碎片，使内存出现不连续的状态，这种状态会对后续的内存分配造成问题，一旦有大的对象需要分配内存，可能出现所有碎片空间都无法分配内存的情况，会提前触发垃圾回收

**标记回收**：在标记清除的基础上，对碎片进行了整理，将活着的对象往一端移动，移动完成后，直接清除掉边界外的内存完成回收

由于标机回收比较耗时，所以主要使用的还是标记清除，当新生代像老生代晋升但空间不足时，才会使用标记回收。

### 用过的nodejs框架（express,koa等），介绍一下中间件的原理

koa：基于node的一个web开发框架，利用co作为底层运行框架，利用Generator的特性，实现“无回调”的异步处理；

koa2的中间件： 1、通过async await实现的，中间件执行的顺序是“洋葱圈”模型。 2、中间件之间通过next函数联系，当一个中间件调用next()后，会将控制权交给下一个中间件，直到下一个中间件不再执行next()后，会沿路返回，将控制权交给前一个中间件。

compose的实现

这里面有一些亮点：

这是一种尾递归的形式，尾递归的特点是最后返回的值是一个递归的函数调用，这样执行完就会在调用栈中销毁，不会占据调用栈.

返回的是一个Promise.resolve包装之后的调用，而不是同步的调用，所以这是一个异步递归，异步递归比同步递归的好处是可以被打断，如果中间有一些优先级更高的微任务，那么可以先执行别的微任务

compose是函数复合，把n个middleware复合成一个，参数依然是context和next，这种复合之后依然是一个middleware，还可以继续进行复合。

### EXPRESS & KOA

**相同点**

express 和 koa 是使用 node.js 编写的 Web 框架

功能上，express 和 koa 都支持使用中间件（middleware）来扩展框架功能

区别

express从nodejs的0.1开始支持，koa从v7.6.0开始后支持

express 自带路由、开发模板引擎等功能，而 koa 自身不包含任何中间件

在 express 的中间件执行是**线性执行**的，就像排队一样，每个中间都需要执行 next 函数通知下一个中间件执行，最后需要显性的调用 res.end 或者 res.send 来完成请求响应，koa 的中间件模型为洋葱型，结合 async/await 语法，无需像 express 一样手动调用 res.end 或 res.send，koa 会在最后一个中间件的 Promise 状态完成后自动响应，也方便在其中做各种操作。

## RPC

### 什么是RPC，介绍一下它的工作原理

**概念** RPC是一种进程间通信方式，主要是为了解决分布式系统中的服务调用问题；RPC是一个C/S架构。

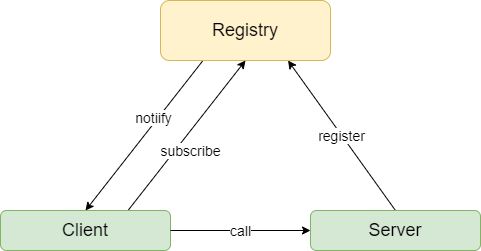
RPC是指远程过程调用，也就是说两台服务器A，B，一个应用部署在A服务器上，想要调用B服务器上应用提供的函数/方法，由于不在一个内存空间，不能直接调用，需要通过网络来表达调用的语义和传达调用的数据

**RPC要解决两个问题**：

1. 解决分布式系统中，服务之间的调用问题。
2. 远程调用时，要能够像本地调用一样方便，让调用者感知不到远程调用的逻辑。

**rpc的架构思想**

在RPC框架中主要有三个角色：Clinet，Server和Registry，如下图所示：



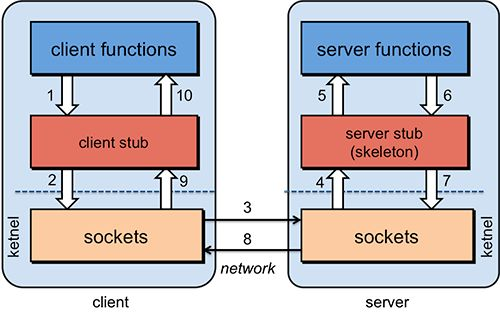
* Server: 暴露服务的服务提供方。
* Client: 调用远程服务的服务消费方。
* Registry: 服务注册与发现的注册中心。

Server端启动后主动向注册中心Registry注册Server端机器的IP、port以及提供的服务列表；Client端启动时向注册中心获取服务提供方地址列表，调用（call）指定的服务。

**如果想要自己实现一个RPC，最简单的方式要实现三个技术点，分别是：**

1. 服务寻址
2. 数据流的序列化和反序列化
3. 网络传输

**rpc的调用过程**

****

1. Client像调用本地服务似的调用远程服务；
2. Client stub接收到调用后，将方法、参数序列化；
3. 客户端通过sockets将消息发送到服务端；
4. Server stub收到消息后进行解码（将消息对象反序列化）；
5. Server stub根据解码结果调用本地的服务；
6. 本地服务执行(对于服务端来说是本地执行)并将结果返回给Server stub；
7. Server stub将返回结果打包成消息（将结果消息对象序列化）；
8. 服务端通过sockets将消息发送到客户端；
9. Client stub接收到结果消息，并进行解码（将结果消息反序列化）；
10. 客户端得到最终结果。

### RPC相对于Http有什么优缺点

<https://www.zhihu.com/question/41609070>

rpc是远端过程调用，其调用协议通常包含**传输协议和序列化协议。**

传输协议包含: http2 协议和tcp协议。

序列化协议包含: 如基于文本编码的 xml json，也有二进制编码的 protobuf hessian等。

**所谓rpc与http的对比，大约是针对自定义tcp协议的rpc和http的对比，因为rpc的实现也可以是基于http2的**

所以对比的点主要是http的tcp协议与我们自定义的tcp协议上的区别

需要否定：

* http协议相较于自定义的tcp报文协议，增加的开销在于链接的建立与断开。因为http2已经支持了连接的复用。
* http也可以使用protobuf这种二进制编码协议对内容进行编码

所以两者最大的区别在于传输的协议上。

http1.1协议的tcp报文包含太多的费信息，请求头的键值对使用了文本编码。报头所占的比列太高，可能要有效字节数的占比还不到一半。而自定义tcp协议，就可以缩减不必要的信息。**极大的精简传输内容。当然这些都是针对http1.1，因为http2.0已经优化了编码效率问题**

http需要可读性强，包括输入、输出，解析等。

  rpc框架是面向服务的更高级的封装。如果把一个http servlet容器上封装一层服务发现和函数代理调用，那它就已经可以做一个rpc框架了。

**总结：**

  RPC主要用于公司内部的服务调用，性能消耗低，传输效率高，服务治理方便。HTTP主要用于对外的异构环境，浏览器接口调用，APP接口调用，第三方接口调用等。

## 数据库相关

### 你知道的mysql存储引擎（一般都是对比MyISAM和InnoDB），各自区别（使用场景）

### 非关系数据库和关系型数据库的区别，各自优缺点

存储方式

非关系型：虚拟内存+持久化

关系型：不同引擎不同存储方式

### 什么是事务

事务就是把多个操作作为一个整体来执行的功能

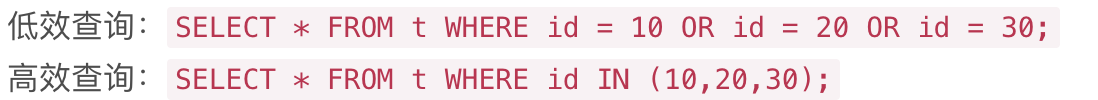
事务的四个特性

* 原子性：将所有sql作为一个原子工作单位执行，要么都执行，要么都不执行
* 一致性：事务完成以后，所有数据的状态都是一致的
* 隔离性：如果有多个事务并行处理，那么事务做出的修改必须与其他事务隔离
* 持久性：事务完成后，对数据库的修改是一个持久化的操作

### 数据库的乐观锁和悲观锁是什么

### 如何进行mysql优化

* 尽量避免使用子查询
* 用in代替or



* 读取适当的记录用limit，不要读取多余的数据
* 禁止不必要的order by排序：如果我们对结果没有排序的要求，就尽量少用排序；如果排序字段没有用到索引，也尽量少用排序
* 总和查询可以禁止排重用union all：union和union all的差异主要是前者需要将结果集合并后再进行唯一性过滤操作，这就会涉及到排序，增加大量的CPU运算，加大资源消耗及延迟。当然，union all的前提条件是两个结果集没有重复数据。所以一般是我们明确知道不会出现重复数据的时候才建议使用 union all 提高速度。
* 避免随机抽取记录
* 将多次插入换成批量插入

### 如何定位和处理慢查询问题

# 网络安全

### 有关编码

**编码方式**

* askii码，主要收录英语和欧洲一些国家的语言，是单字节编码
* Unicode编码，可以有很多不同的编码方式，旨在收录全球所有字符
* utf-16是Unicode的一种使用方式，为双字节编码
* utf-8是Unicode的一种使用方式，为传输而设计的编码，采用变长的编码方式，可使用1~4个字节来表示一个字符
* gbk是最早的一版汉字编码，注意的是UTF-8一个中文字符占3个字节）

**前端中的编码**

* base64
* URL编码规则：需要编码的字符转换为UTF-8编码，然后在每个字节前面加上%。
* encodeURI：encodeURI不编码的82个字符：!#$&’()\*+,/:;=?@-.\_~0-9a-zA-Z，从中可以看不会对url中的保留字符进行编码，所以适合url整体编码encodeURIComponent：对url中的保留字进行的编码，所以当传递的参数中包含这些url中的保留字（@，&,=)，就可以通过这个方法编码后传输

### 什么是XSS攻击，它有哪几种，分别是怎么实现的，以及如何防范

攻击者通过网站注入点注入恶意客户端可执行的脚本，当用户访问网站时，可执行的恶意脚本通过网站执行点执行来达到某些目的，比如获取用户权限，恶意传播，钓鱼

1. **常见的xss分类**
2. 存储型 XSS
   1. 攻击者将恶意代码提交到目标网站的数据库中。
   2. 用户打开目标网站时，网站服务端将恶意代码从数据库取出，拼接在 HTML 中返回给浏览器。
   3. 用户浏览器接收到响应后解析执行，混在其中的恶意代码也被执行。

这种攻击常见于公开用户提交数据（UGC 内容）的网站功能，如论坛发帖、商品评论、用户私信等。

1. 反射型 XSS

反射型 XSS 漏洞常见于通过 URL 传递参数的功能。

由于需要用户主动打开恶意的 URL 才能生效，攻击者往往会结合多种手段诱导用户点击。

* 1. 攻击者构造出特殊的 URL，其中包含恶意代码。
  2. 用户打开带有恶意代码的 URL 时，网站服务端将恶意代码从 URL 中取出，拼接在 HTML 中返回给浏览器。
  3. 用户浏览器接收到响应后解析执行，混在其中的恶意代码也被执行。

反射型 XSS 漏洞常见于通过 URL 传递参数的功能，如网站搜索、跳转等。由于需要用户主动打开恶意的 URL 才能生效，攻击者往往会结合多种手段诱导用户点击。

1. DOM 型 XSS
   1. 攻击者构造出特殊的 URL，其中包含恶意代码。
   2. 用户打开带有恶意代码的 URL。
   3. 用户浏览器接收到响应后解析执行，前端 JavaScript 取出 URL 中的恶意代码并执行。

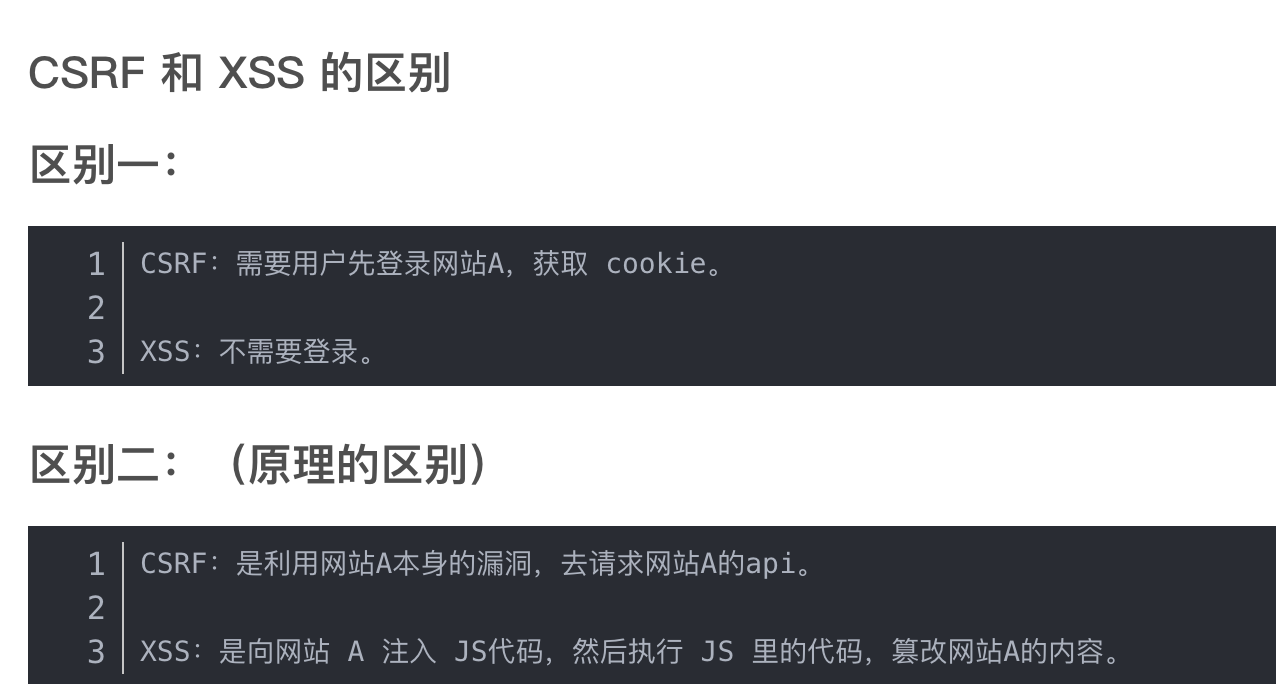
DOM 型 XSS 跟前两种 XSS 的区别：DOM 型 XSS 攻击中，取出和执行恶意代码由浏览器端完成，属于前端 JavaScript 自身的安全漏洞，而其他两种 XSS 都属于服务端的安全漏洞。

1. **防止xss攻击**

<https://segmentfault.com/a/1190000022984648>

* 1. 对文本内容做充分的转义
  2. 针对富文本内容进行白名单过滤
  3. 不要使用innerHtml()/outerHtml()/document.write()等方法对后台的数据做处理
  4. 使用csp来（禁止加载外域代码；禁止内敛脚本执行；禁止外域提交，网站被攻击后，用户的数据不会泄露到外域）
  5. 使用http-only cookie来禁止js读取某些铭感的cookie，攻击者完成xss注入后也无法窃取此cookie
  6. 使用验证码，来防止脚本冒充用户提交危险操作

### 什么是CSRF攻击，它与XSS攻击的区别，如何防范



#### csrf背景

scrf,简称跨站请求伪造。

由于http是无状态的，每次服务器都需要收到请求参数后才能确定用户的行为，服务器并不清楚在请求前发生了什么，所以这是一个无状态协议。

但是在现实情况下很多都需要有状态，比如用户态，因为网页上很多的操作都是需要登录以后才能判断是否能够操作，那如何去判断这个用户态呢，主要就是http请求的时候都把用户态的信息带过去传给后台服务器，后台服务再根据带过来的信息判断用户合法性再进行各种操作。

那让每次http请求都带上用户信息的方式就是cookie。

**cookie的特征：**

* 浏览器自动携带本次域名请求的cookie
* 读写cookie有跨域的限制
* 有生命周期

#### 攻击过程

因为登录态cookie的信息是浏览器默认携带的，用户访问了网站a，只要浏览器不关闭，这个网站a产生的cookie就会一直存在，此时用户去访问一个黑客网站b，黑客网站b发送http请求到网站a，那浏览器就会自动给他带上用户态cookie信息，就可以模拟用户进行一系列增删改查的操作

读操作是否会被攻击到呢？默认大部分情况下是拿不到的，除非以下两种情况：

* 后台返回的数据是jsonp格式的，并且是get请求，此时黑客可以拿到数据
* 后台通过cors处理了跨域，并且没有对请求图的origin做来源的限制

#### csrf防护

防护方式很简单，能够识别请求源是来自用户的页面还是黑客的页面就可以

1. referrer

referrer是浏览器默认带上的，含义是发送请求的网站的地址，可识别发送请求的地址是不是白名单预期的地址

1.1 referrer会不会被伪造或者篡改

1. 在浏览器环境下，referrer字段是浏览器自带的，注入的js脚本是无法修改该字段的
2. 浏览器自带的插件可以修改referrer字段，但这不属于csrf的攻击范畴，但是如果都已经在浏览器上装上了插件，那就有更简单的方式来进行攻击
3. 通过网关或者抓包的方式可以修改refferer，但是这个是中间人攻击，需要使用https来解决
4. 黑客通过自己后台代理，把请求先发到自己的后台服务，然后修改refferer再转发到需要攻击的后台，这种方式拿不到cookie，所以行不通

1.2 referrer防csrf是安全的吗

处理好以下几种情况，referrer是安全的

1. 读操作：读操作不能用jsonp，cors要设置白名单，这样读操作就安全
2. 写操作：referrer要求不能为空，使用白名单。不能用get请求，比如用户写日志可自定义输入图片链接，而图片的img的src或者a标签的href就指向了写操作的url，这样只要打开这篇日志，就会发送get请求，此时的请求就自动带有referrer

什么情况下referrer为空？

地址栏直接输入url的时候，第一次请求referrer为空，一般是html页面，这种读操作不用防csrf

1. referrer-policy设置no-referrer时
2. 一些iframe的特殊使用也可能导致为空
3. 用origin可以吗

可以，和referrer原理一样，他是xhr2.0增加的，含义是发送请求页面的域名，主要是为了解决跨域问题。用他同样要设置白名单，不允许为空。

还可以自定义http请求头，升级为复杂请求，这样跨域的情况下会先发一个option请求预检

1. token

上面讲到Cookie的一些特性的第二条，读写Cookie有跨域限制(作用域，Domain,Path)，所以我们可以用这个特性来区分是自己页面还是黑客页面。只要页面能读（或者写）[www.photo.com](http://www.photo.com/)域名Cookie,就证明是自己的页面。懂了原理，方案就很简单，比如服务器通过cookie下发一个token，token值是随机数，页面发请求的时候从cookie取出token通过HTTP请求参数传给后台，后台比对参数里的token和cookie里的token是否一致，如果一致就证明是自己页面发的请求，如果不一致就返回失败。防CSRF的方案就是这么简单，这种方法能够100%防CSRF，但是可能会有几个变种，下面探讨几种情况

1. Token放在HTTP参数里的哪里？

放在URL的querystring里，Post请求的Data里或者HTTP请求头里，这三种方式都可以，只是有一点点细微的差别，如果querystring里，可能会影响Get请求的缓存效果，因为重新登录之后token会变，url也就变了，之前的缓存就失效了。如果放在HTTP请求头里，就需要使用fetch或者XHR发请求，这样会变成复杂请求，跨域情况需要多一次Option预检请求，对性能多少有一点点影响。

1. 用Token方案后写操作可以用Get么？

可以，从安全角度考虑是可以的，用了Token之后，Get和Post的安全等级是一样的。上面讨论的那种少见的CSRF攻击过程也攻击不到了。但是从语义化考虑建议Get是还是处理读操作方便理解。

1. 用Token方案后读操作可以用JSONP跨域么？

可以，可以使用JSONP跨域了，另外如果使用CORS处理跨域，建议还是需要对请求头的Origin做白名单限制，防止不同子域名相互影响。

### SQL注入

SQL 注入就是在用户输入的内容被作为 SQL 语句的一部分被执行，而攻击者利用这一漏洞输入非法内容，使服务端执行了计划外的 sql操作，导致一些未授权的操作被执行。

不能相信用户的任何输入，如果一个 sql 语句里包含了用户输入的内容，那我们要对内容做 sql 安全相关的过滤检查。同时，不使用拼凑字符串型的语句执行方式，改用占位符预处理的参数化语句

### 中间人劫持

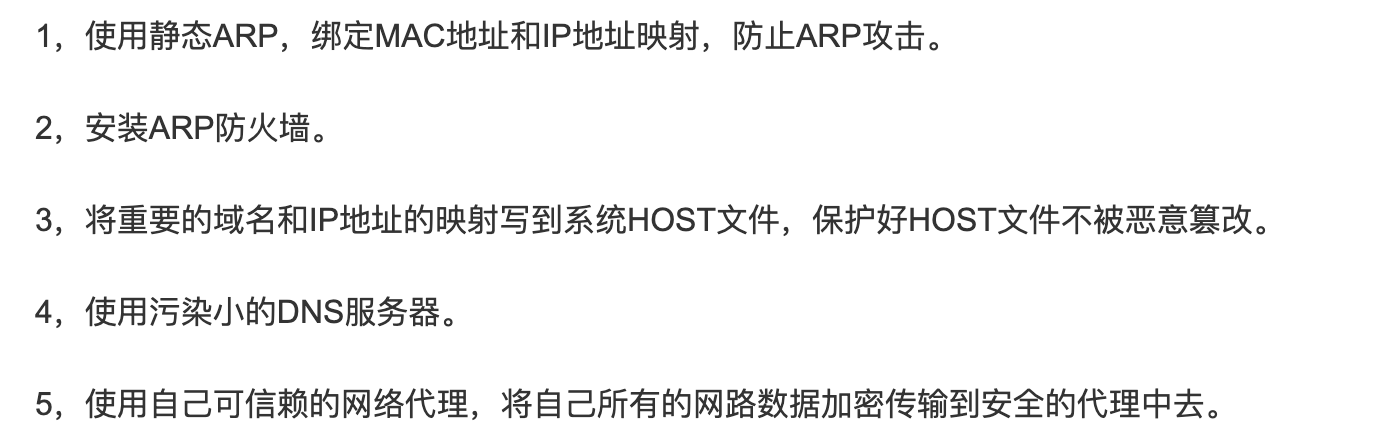
（英语：Man-in-the-middle attack，缩写：MITM）

攻击者与通信双端分别建立独立的连接，并交换其所收到的数据，使通信双端认为他们正处在一个私密的链接中通信，但事实上整个会话已经被攻击者完全控制

常见的是dns劫持，目标将其dns请求发送到了攻击者伪造的dns服务上，攻击者伪造dns响应，替换正确的ip，之后登录的就是攻击者指定的ip，而该ip上攻击者就可以实行一系列攻击

http劫持：就是在用户和服务器的信息传输之间加入其他信息，比如广告。用户发送http请求，运营商的路由器收到请求，识别到为http请求，路由器的旁路设备标记此次tcp链接为http协议后，直接返回修改后端html代码，导致浏览器被插入广告

**预防措施：**



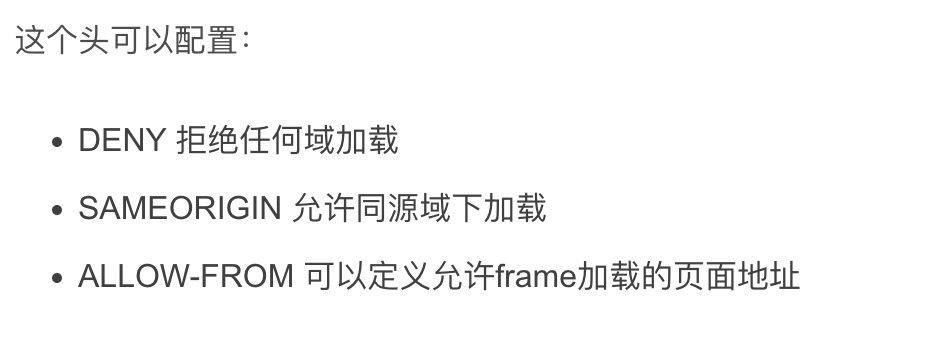
### 点击劫持

通过在网页的控件上覆盖不可见的框，使用户误认为在操作可见性的空间，从而实现攻击

主要利用css中的opacity、position、z-index。以及iframe）

防范：

* 利用js代码，禁止iframe嵌套（frame busting）
* 设置http头x-frame-options为DENY。来防御利用frame嵌套的点击劫持攻击



* csp也可以防止点劫持

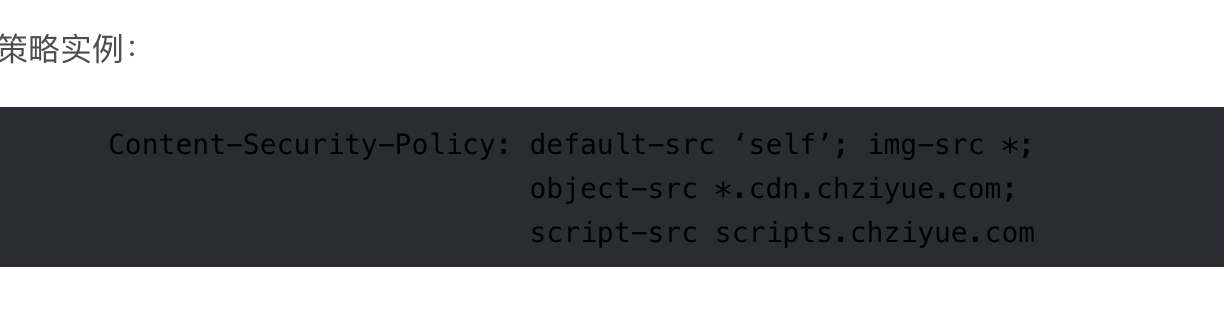
### CSP原理了解一下

内容安全策略（Content Security Policy）用于帮忙检测和缓解xss的攻击

csp的本质更像是白名单制度，开发者明确告诉客户端，哪些资源可以加载哪些资源不可以加载，等同于提供白名单。它的实现和执行全部由浏览器完成，开发者只需要提供配置

通过server返回conntent-sercurity-policy头来启动不同程度的防护措施，，CSP可以完全禁止内联的JavaScript，并且控制外部代码从哪里加载。它也可以禁止动态代码执行。禁用了所有的攻击源，XSS攻击变得更加困难。

实现上可以在meta标签也可以在header头



该策略默认只允许资源从同一来源加载，但允许图像从任何URI加载，插件内容只从指定的CDN地址加载，外部脚本仅从scripts.chziyue.com加载。

**策略测试**

有关CSP的一个好处是，能够在执行一个策略的同时并行测试其他策略。这意味着，可以在比开发环境要复杂得多的生产环境中部署测试策略。

Content-Security-Policy-Report-Only响应头用于创建一个仅用于测试的策略：

Content-Security-Policy-Report-Only: default-src ‘self’

如果只启用报告的策略出错，并不会阻塞请求，但可以配置报告使得故障可被传递回起初的网站。

## TS

### 怎么理解ts

<https://www.zhihu.com/question/335339124>

**优点**

ts作为js的超集，可以对js进行静态类型检查，可以编写更健壮、更可维护的大型项目。

TypeScript 要编译后才能使用。所以我们的类型错误会在编译过程中被编译器发现，更早发现 BUG。

目前在绝大多数 IDE（集成开发环境）中已经支持 TypeScript 的 智能提示、自动补全、代码导航 等功能，

**缺点**

需要写更多代码、有一定的学习成本、多了一步编译的操作

### ts为什么又叫anyscript

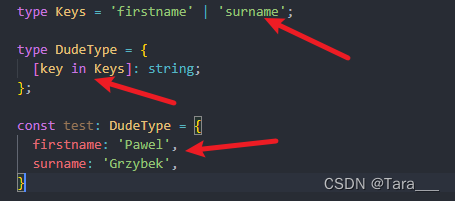
因为需要兼容 JavaScript 的缘故，TypeScript 的类型是可选的。你可以用 any 类型，也可以进行类型的强制转换，所以如果你在代码中写了太多这样的东西，甚至将其变成 AnyScript。那么实际运行还是可能会出现一些类型上的问题。

### interface和type的区别

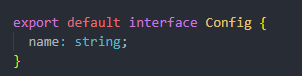
1. typeo的类型别名可以用于其他的类型，比如 联合类型、元组类型、基本类型，interface 不行。

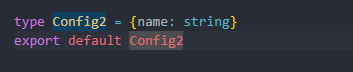


1. type 的别名不可以多次定义会报错，而 interface 则可以多次定义，会将其视为合并到一起。
2. type 能用 in 关键字，而interface不行。



1. 默认导出的方式不同，inerface 支持同时声明，默认导出，而type必须先声明后导出





5.扩展方式不同，interface 用 extends 来扩展，而type 用 & 操作符扩展

## 拓展

### web worker & service worker & worklet

<https://mp.weixin.qq.com/s/3Ep5pJULvP7WHJvVJNDV-g>

概括地说，Web Worker，Service Worker和Worklet都是在与浏览器页面线程不同的线程上运行的脚本。它们的不同之处在于它们的使用位置以及启用这些用例所必须具备的功能。

* worklet 是worker的轻量级版本，与浏览器的渲染管道挂钩，使我们能够访问到渲染管道的底层部分
* service worker本质上就是在web worker的基础上增加了离线缓存的能力。（因为本身web worker是临时的，结果不能被持久存下来）是浏览器和网络之间的代理。通过拦截网站发出的请求，service worker可以将请求重定向到缓存，从而实现脱机访问。
* Web Worker一个独立于JavaScript主线程的独立线程，在里面执行需要消耗大量资源的操作不会堵塞主线程。

### webassembly

WebAssembly 允许网站用诸如 Rust、C/C++、Java、Python 等编程语言编写代码，然后编译成字节码格式，实现在 Web 浏览器在运行。

较成功的案例：figma

优点：适合计算密集型应用，并且他的执行速度比没有优化的js快

但是：

* WASM 不是计算密集型应用的唯一救星。对于真正高度并行且精度要求不高的算法，用 WebGL 做 GPU 加速往往更快，然后才是 WASM 这种 CPU 上运行的字节码。
* 对于极致优化后的js，wasm并没有快多少
* 前端框架应用的场景是io密集型的，切换语言的投入产出比很难说

### beacon

信标（Beacon）可以异步与非阻塞的数据传输，从而最大限度地减少与其他关键操作的资源争用，同时它可以确保这些请求一定会被处理并将其传递到服务端：

* 信标请求优先避免与关键操作和更高优先级的网络请求竞争。
* 信标请求可以有效地合并，以优化移动设备上的能量使用。
* 保证页面卸载之前启动信标请求，并允许运行完成且不会阻塞请求或阻塞处理用户交互事件的任务。

### SSG (静态页面生成)&ISR(增量页面渲染)

ssg：对于一些页面，每次渲染出的大部分内容都是一样的，这一部分内容就可以使用ssg，传统页面请求到服务端以后服务端进行渲染再返回，而ssg就是提前把渲染好的页面缓存起来（即提前进行一个预渲染的工作），请求过来时这接把页面返回

isr：对于需要渲染的页面量级很大的情况，不可能对所有页面进行一个预渲染，所以只会对关键的页面进行预渲染再缓存，用户如果请求的是非关键页面，就会先拿到fallback页面，然后进行csr，同时服务进行异步的预渲染，之后存到缓存，提升用户的体验

### bun

与node、deno一样，是js运行时。

 Node 和 Deno 用的 JS 引擎是 V8 (Chromium)，Bun 用的是 JavaScriptCore (Safari)，不同的引擎也有可能产生性能差异。

bun与node的性能对比中。无论是服务运行、sql处理、调接口，性能上都是node的三倍左右

官方还称其为： all-in-one JavaScript runtime；

所谓 all in one，是因为 Bun 和传统的 Node.js 这种传统的 javaScript 运行时 不同； Bun 提供了 打包、转译、安装和运行 JavaScript & TypeScript 项目的功能，内置原生打包器 (native bundler)、转译器、task runner、npm 客户端，以及数百个 Node.js API 和 Web API，包括约 90% 的 Node-API 函数（本机模块）、fs、path、Buffer 等。

这就意味着你可以实现在 Bun 上运行的大部分 Node API 和 Web API， 还可以在不需要使用 Webpack、Rollup、esbuild、babel 等工具的前提下直接运行 TypeScript 、JSX 了；

## 数据结构

### 链表优缺点

**链表的优点如下**：

1. 链表能灵活地分配内存空间；
2. 能在 O(1) 时间内删除或者添加元素，前提是该元素的前一个元素已知，当然也取决于是单链表还是双链表，在双链表中，如果已知该元素的后一个元素，同样可以在 O(1) 时间内删除或者添加该元素。

**链表的缺点是：**

1. 不像数组能通过下标迅速读取元素，每次都要从链表头开始一个一个读取；
2. 查询第 k 个元素需要 O(k) 时间。

## 算法

分治算法、动态规划算法、贪心算法、回溯法、分治界限法

1. 首屏整个过程：【dns解析、tcp链接】200多ms =》 发送请求包 =》 服务端响应（90ms）=》 网络回包 =》html渲染过程（200ms）

【存在的问题】：资源包过大、css使用率低

1. 页面搭建、技术选型、遇到问题、提升

由于团队使用react+koa，所以这个项目的搭建是使用的react、服务端使用的是koa。由于这个项目需要承载多个业务，所以使用react路由在controller层统一做页面的ssr，在service层再对具体的业务，然后再data层完成具体数据的处理

**以及整个项目针对同构直出做了一些工作，包括：**

1. 首先是渲染层面上，node调用react ssr api（renderToString）进行组件渲染并注入到html返回，在浏览器第一次渲染的时候调用ReactDOM.hydrate进行注水，得到一个可交互页面

2.数据层面上，保证数据一致性，react项目我们需要进行状态管理，服务端得到的数据，在返回的时候注入script标签，在浏览器初始化全局store的时候使用该数据，完整前后端渲染数据的共享。

3. 项目使用react router管理多页面，所以需要保证路由一致性，在node端和浏览器端分别使用了staticRouter和broswerRouter进行路由管理。

4. 针对执行环境（node和浏览器）的不同，利用打包工具分别构建对应环境的资源bundle。针对像window之类的变量和api在node端并没有情况，代码里也针对执行环境不同进行了不同的处理，比如自定义方法判断环境等。