前端学习笔记

1. 执行上下文

定义：简而言之，执行上下文评估和执行javascript代码环境的抽象概念；

类型：全局执行上下文、函数执行上下文、eval上下文；

执行栈：调用栈，一种拥有LIFO（先进后出）数据结构的栈，被用来存储代码执行时创建的所有上下文；

创建上下文：两个阶段

1. 创建阶段
   1. this值的决定，即this绑定；
   2. 创建词法组件环境；
      1. 词法环境定义：简单来说词法环境是一种持有标识符—变量映射的结构。（这里的标识符指的是变量/函数的名字，而变量是对实际对象[包含函数类型对象]或原始数据的引用）；
      2. 组成：环境记录器（存储变量和函数声明的实际位置）、一个外部环境的引用（意味着它可以访问其父级词法环境（作用域））；
   3. 创建变量环境变量
   4. 不同：在 ES6 中，词法环境组件和变量环境的一个不同就是前者被用来存储函数声明和变量（let 和 const）绑定，而后者只用来存储 var 变量绑定。
2. 执行阶段：完成对所有这些变量的分配，最后执行代码。
3. 作用域

定义：程序中定义变量的区域，它规定了如何查找变量，也就是确定了当前执行的代码对变量的访问权限

词法作用域：静态作用域，函数的作用域在函数定义的时候就已经决定了。javascript采用的就是词法作用域；

动态作用域：函数的作用域是在函数执行时才决定的，如.bash文件执行；

代码示例1：

var value = 1;

function foo() {

console.log(value);

}

function bar() {

var value = 2;

foo();

}

bar();

打印结果：1；因为js是词法作用域，定义时就已经决定了：函数执行时，函数内部没有value，查找到定义位置的上一级，所以等于1，而不是2；

分类：

1. 全局作用域
2. 函数作用域：一般情况下，函数之前的作用域独立，不能互相访问；
3. 块级作用域：es6引用，使用let和const创建的变量拥有块级作用域

暂时性死区：块级作用域；

示例1：

function foo() {

console.log(bar)

let bar = 3

}

foo() // 会报错：Uncaught ReferenceError: bar is not defined。括号 { 与变量生命前的区域就是暂时性死区；

示例2：

function foo(arg1 = arg2, arg2) {

console.log(`${arg1} ${arg2}`)

}

foo(undefined, 'arg2')

// Uncaught ReferenceError: arg2 is not defined 当第一个参数缺省时，执行 arg1 = arg2 会当作暂时性死区处理：

注：第一个参数换成null就不会报错

作用域链：变量查找时，优先在当前作用域进行查找，当找不到时，就会向上一层作用域查找，直到查到到全局作用域，这样作用域之间就形成一条链，就是作用域链

1. JS基础

数据类型：7种基础类型：null、undefined、number、string、symbol、boolean、 bigint；引用数据类型：Object；区别：基础类型存储在栈内存， 被引用或拷贝时，会创建一个完全相等的变量；引用类型存 储在堆内存，存储的是地址，多个引用指向同一个地址，这 里会涉及一个“共享”的概念。

数据类型检测：

Typeof： 除null之外的简单数据类型，引用数据类型除了function， 其他都是object，null也是object；

Instanceof：判断左边是否为右边构造函数生成的实例；

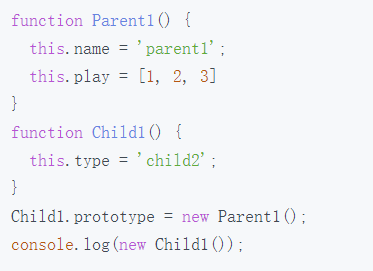
代码实现原理：逐级查找左边的原型，是否等于右边的prototype



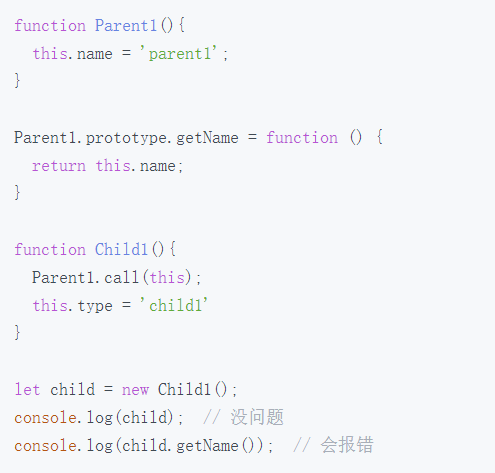
Object.prototype.toString:目前来说，最准确的方法，不过在ie6下会 有兼容性问题，而且还需要后续提取操作，性能开销大；

1. 继承

原型链继承：将子类的原型指向父类的实例，缺点：子类间共用父类内存空间，一个改变会影响另一个



构造函数继承（借助 call）：它使父类的引用属性不会被共享，优化了第一种继承方式的弊端；但是随之而来的缺点也比较明显——只能继承父类的实例属性和方法，不能继承原型属性或者方法。



组合继承（前两种组合）：父类被构造执行两次，增加了性能开销



原型式继承：Object.create，多个实例的引用类型属性指向相同的内存，存在篡改的可能



寄生式继承：原型式继承可以获得一份目标对象的浅拷贝，然后利用这个浅拷贝的能力再进行增强，添加一些方法



寄生组合式继承：相对最优的继承方式



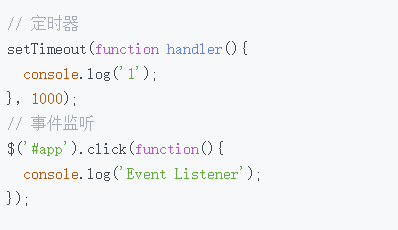
1. 闭包

定义：闭包其实就是一个可以访问其他函数内部变量的函数。即一个定义在函数内部的函数，或者直接说闭包是个内嵌函数也可以；

产生本质：当前环境中存在指向父级作用域的引用；

场景：

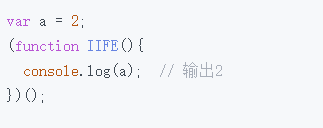
1. 返回一个函数；
2. 在定时器、事件监听、Ajax 请求、Web Workers 或者任何异步中，只要使用了回调函数，实际上就是在使用闭包。



1. 作为函数参数传递的形式



1. IIFE（立即执行函数），创建了闭包，保存了全局作用域（window）和当前函数的作用域，因此可以输出全局的变量



作用：封装具有私有变量；

弊端：内存泄漏，用完及时清除

1. this

定义：是在调用函数执行时根据上下文决定的

规则：

1. 简单调用函数时，严格模式是指向undefined，否则指向window/global；
2. 利用new构造函数执行，绑定到新创建的对象上；
3. bind/call/apply显示调用，绑定到指向参数的对象上；

注：非严格模式下，如绑定值为null和undefined，会自动绑定到window，严格模式，制定值本身；如为其他简单数据类型，则指向该原始值自动包装对象

1. 箭头函数没有自己的this，根据外层上下文绑定this；

箭头函数和普通函数的重要区别:

1. 没有自己的this.super\arguments\和new。target绑定
2. 不能使用new来调用；
3. 没有原型对象；
4. 不可以改变this的绑定；
5. 形参名称不能重复

代码示例：

var student = {

name: '若川',

doSth: function(){

console.log(this.name);

return () => {

console.log('arrowFn:', this.name);

}

}

}

var person = {

name: 'person',

}

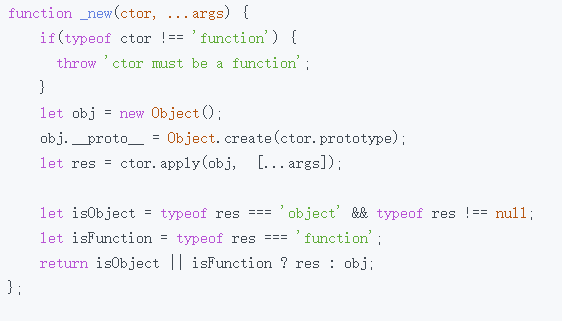
student.doSth().call(person); // '若川' 'arrowFn:' '若川'

student.doSth.call(person)(); // 'person' 'arrowFn:' 'person'

1. 一般上下文的对象调用，this指向该对象；

new的过程：

1. 创建一个新的对象；
2. 这个对象会被执行原型链接；
3. 生成的新对象会绑定到函数调用的this；
4. 通过new创建的每个对象都最终被[[prototype]]链接到这个函数的 prototype对象上；
5. 如果函数没有返回对象类型Object(Function, Array, Date, RegExg), 那么new表达式中的函数调用会自动返回这个新的对象: new 关键词执行之后总是会返回一个对象，要么是实例对象，要么是 return 语句指定的对象。
6. 代码示例：



Apply实现：就是把调用函数添加到目标对象中去

Function.prototype.myApply = function(context) {

Var context = context || window

Let fn = Symbol();

Context[fn] = this

Var args = arguments[1]

If( args === undefined ) {

Context[fn]()

}else {

Context[fn](…args)

}

Delete context.fn

}

Apply/call/bind总结:

1. 三者都可以改变this指向，指向第一个参数；
2. Apply/call会立即执行函数，bind是返回一个函数，可待后续执行；
3. Call的参数需要逐个列出，apply是一个数组
4. 拷贝

浅拷贝：只是复制值的引用，而非复制真正的值；

方法：

1. ...展开运算符；
2. Object.assign:不会拷贝对象的继承属性\不会拷贝对象的不可 枚举的属性\可以拷贝 Symbol 类型的属性
3. Concat
4. Slice

深拷贝：真正复制值，复制后改变值不会影响原有值；

深拷贝方法：

1. JSON.stringify/parse:undefined、function、Symbol在转化时会被 忽略\拷贝 Date 引用类型会变成字符串\无法拷贝不可枚举的属 性\无法拷贝对象的原型链\拷贝 RegExp 引用类型会变成空对 象\对象中含有 NaN、Infinity 以及 -Infinity，JSON 序列化的结果 会变成 null；
2. 递归拷贝复制：

Function clone(x) {

// 如果不是对象或者数组，直接返回

If( typeof x !== ‘object ) return x

// 如果是数组

Let res;

If( x instanceof Array ) {

For(let i=0;i<x.length; i++) {

// 避免一层死循环 a.b = a

Res[i] = x[i] === x ? res : clone[x]

}

}

If() { // 如果是object

For(let key in x) {

If(x.hasWonProperty(key)) {

// 避免一层死循环 a.b = a

Res[key] = x[key] === x ? res : clone[x]

}

}

}

Return res

}

1. 循环克隆：

Function cloneLoop(x) {

Const t = type(x) // 一个类型判断函数

Let root = x;

If(t = ‘array’’ ) {

Root = []

}else if(t === ‘object’) {

Root = {}

}

Const loopList = [

{

Parent: node,

Key: undefined,

Data: x

}

]

While(loopList.length) {

// 深度优先

Const node = loopList.pop();

Const parent = node.parent;

Const key = node.key;

Const data = node.data;

Const tt = type(data)

// 初始化赋值目标，key为undefined则复制到赋元素，否则复制到子元素

Let res = parent;

If(typeof key !== ‘undefined’) {

Res = parent[key] = tt === ‘array’ ? [] : {}

}

If(tt === ‘array’) {

For( let i = 0; i<data.length; i++ ) {

If(data[i] === data) {

Res[i] = res

}else if(isClone(data[i])) { // 是数组或者对象

// 下一次循环

loopList.push({

Parent: res,

Key: i,

Data: data[i]

})

}else {

Res[i] = data[i]

}

}

}

else if( tt === ‘object’) {

For(let k in data) {

If(hasOwnProperty(data, k)) {

// 循环引用

If(data[k] === data) {

Res[k] = res

}else if(isClone(data[k])) {

// 下一次循环

loopList.push({

Parent: res,

Key: i,

Data: data[k]

})

}else {

Res[k] = data[k]

}

}

}

}

}

Return root

}

1. Eventloop

定义：event loop是让js做到即是单线程，又绝对不会阻塞的核心机制，也是 js并发模型的基础，是用来协调各种事件、用户交互、脚本执行、UI渲染、 网络请求等的一种机制

执行&运行：

执行：一般执行依赖与环境，比如node/浏览器等，js在不同环境的执行 机制可能不相同；

运行：指的是js的解释引擎，这是统一的；

什么是EventLoop：

1. 主线程至上而下执行所有代码；
2. 同步任务直接进入主线程执行，异步任务则进入event table 并注册 回调函数；
3. 异步任务完成后，event table会将这个回调函数移入event queue；
4. 主线程任务执行完毕后， 会从event queue中读取任务，进入到主线 程执行；
5. 在这个event queue中又会分宏任务和微任务两种，先完成微任务队 列再去执行宏任务队列
6. 上述动作会不断轮询，就形成了eventloop；



浏览器event loop与nodejs环境的区别：

（1）浏览器的微任务队列是在每个宏任务队列执行完之后执行；nodejs会 在事件循环的各个阶段之间执行，也就是一个阶段执行完毕，就会去 执行微任务队列；

（2）举例：有两个settimeout代码里面嵌套promise微任务，在浏览 器里面 的话，是逐步执行完每一个settimeout的所有任务，才会执行 下一个 settimeout的代码；在nodejs，就会按照队列类型来执行（同 步->微 任务->宏任务）

1. 函数式编程

定义：是一种编程规范。比起命令式编程，更注重执行的结果而非执行的过程，倡导利用若干简单的执行单元让计算结果不断推进，而不是直接设计一个复杂的执行过程；

特点：

1. 函数是一等公民：函数跟其他数据类型一样，可以作为其他函数 的参数进行输入；
2. 数据不可变：要求所有的数据都是不可变的，如果你想修改一个 对象，应该是新建一个对象，而不是在原有的对象上进行修改；
3. 引用透明：函数不依赖外部的变量；
4. 纯函数：就是没有副作用的函数，相同的输入，相同的输出，不 修改外部数据
5. 惰性计算：函数只在需求时执行，不产生无意义的中间变量；

好处：

1. 代码简洁、开发快速：函数复用高；
2. 接近自然语言：易于理解：使用大量声明式代码，没有乱七八槽 的嵌套循环；
3. 更少出错率：相同的输入，相同的输出，便于测试调试；

缺陷：

1. 性能：一些方法的过度包装，产生一些上下文切换的开销。而且 js本身不属于函数式语言，函数式比直接的指令语句要慢，因为需要 引擎对指令做优化，如map就比纯粹的循环慢；
2. 资源占用：数据不可变，所有需要新建对象，导致垃圾回收压力 大；
3. Web worker

介绍：js一直是单线程环境，我们无法同时运行连个js脚本；html5新规 范中，实现了web worker来引入多线程技术，能让我们在页面主运行 的hs线程加载运行单独的一个或者多个js线程，用来处理一些复杂的 运算，而不会影响主线程UI渲染等，从而可以提高性能效率。但是这 不意味改变js本身单线程的性质，这种事浏览器提供的一个api/能力；

分类：

1. 专用线程：仅能被创建它的脚本使用；
2. 共享线程：能够在不同的脚本使用，一个共享线程对应多个主线 程；

创建：

Var worker = new worker(‘./worker.js’)

注意：加载的worker脚本跟当前创建的脚本需同域；

通信：

1. 传递信息： postMessage
2. 接受信息： onmessage
3. 举例：

Main.js:

var worker = new Worker('worker.js')

worker.onmessage = function(e) {

console.log(e)

}

Worker.js:

console.log('123')

self.postMessage('heelo work')

终止：worker.terminate();

错误监控： worker.onerroe = function(err) {};

环境与作用域：

没有window全局对象，无法访问DOM，可以通过self来访问当前worker实例，一般只能执行纯js运算；

可以使用的部门浏览器api：

setTimeout/setInerval等；

XMLHttpRequest

Navigator

Location

worker中加载外部脚本：使用 importScripts函数



Subworker： worker中的worker：一个worker中再去创建一个worker，目前部 分浏览器还没实现；

共享worker：

New ShareWorker();

应用：

1. 对于图像、视频、饮品的解析处理；
2. canvas中的图像计算处理；
3. 大量的ajax情趣或者网络服务轮询；
4. 大量数据计算处理（排序、检索、过滤、分析）；
5. Vue

mvvm原理：数据劫持+发布订阅模式

数据劫持：通过Object.defineProperty()递归对data的数据每一层都添加上get 和set：oberve函数；

数据编译：complie函数：正则匹配到关键数据插值，然后替换成真实数据，利用replace递归遍历节点，替换成真实数据

发布订阅：当获取数据自动调用get方法时，执行addSubs方法将watcher添加到订阅事件中，当修改数据时，自动调用set方法，执行notify方法将watcher中的update方法执行；

双向数据绑定：v-model: 利用 :value与@input组合的语法糖；

总结步骤：

1. 通过Object.defineProperty的get和set进行数据劫持；
2. 通过遍历data数据进行数据代理到this上；
3. 通过{{}} 对数据进行编译；
4. 通过发布订阅模式进行数据与视图的同步更新；

nextTick：

异步队列：Vue异步执行DOM更新。只要观察到数据变化，Vue将开启一个队列，并缓冲在同一事件循环中发生的所有数据变化。如果一个watcher被触发多次，只会被推入队列一次。这种在缓冲中去除重复数据对于避免不必要的计算和DOM操作非常重要。然后，在下一次事件循环的tick中，Vue刷新队列并执行实际工作。Vue在异步队列中的实现使用原生的Promise.then以及MessageChannel，如果执行环境不支持，会采用setTimeout来代替；

Object.defineProperty Vs Proxy：

1. Object.defineProperty只能劫持对象的属性：需要遍历每一个属性，如果属性也是对象，需要深度遍历，proxy是直接代理对象：不需要遍历操作；
2. Object.defineProperty只能监听原有的属性，无法监听新增属性，需要调用$set（observer），proxy不需要；
3. Proxy对数组监听支持也友好，不需要改写数组方法

Virtual DOM：

定义：是对DOM的抽象，本质上是javascript对象；

为什么需要：

1. 框架在patch过程中尽可能一次性将差异更新到DOM中，而不是更 新整个试图减少DOM的频繁操作；
2. 多人开发过程中在代码层面无需过多关注DOM操作，可以大大提高 开发效率，同时避免因为代码不严谨，过多操作DOM降低性能；
3. Virtual DOM最初的目的，就是为了更好的跨平台，转换成js对象， 再转换成其他原生组件等容易实现，或者SSR等；

关键要素：参照snabbdom.js

Type： 类型，div；

Key： vnode唯一的id；

Data： 包括属性，事件等

Children：子vnode

Text：文本

Elm： 对应的dom

DOM diff 节点对比：

1. 只会在同层级进行比较，不会跨层级比较；
2. 旧节点不存在，插入新节点，新节点不存在，删除旧节点；
3. 新旧节点都是VNode，且新旧节点tag相同

A：对比新旧节点的属性；

B：对比新旧节点的子节点差异，通过key值进行重排序，key值相同 节点继续往下便利

1. 新旧节点如果都是VText，怕大怒你两者文本是否发生变化；
2. 其他情况直接新节点代替旧节点

1. React

React是什么：一个UI框架，通过组件化的方式解决视图层开发复用的问题，本质是一个组件化框架。

特点：

1. 声明式：优势在于直观与组合；
2. 组件化：优势在于视图得拆分与模块的复用，可以更容易做到高内聚低耦合
3. 通用性：一次学习，随处编写。比如React Native，React VR等，通过虚拟DOM的实现。使得react的适用范围很广；
4. 作为一个视图框架，它的劣势也很明显：没有提供一揽子的解决方案，在开发大型应用时，需要整合社区的解决方案。虽然壮大了社区 ，但是让开发者在技术选型上和学习上增加成本；

React为什么使用JSX：

JSX 是一个 JavaScript 的语法扩展，结构类似 XML。 JSX用于react组件声明，但是react不强制使用JSX。即使使用了JSX，在编译过程中也会使用babel插件编译成React.createElement，所以，JSX更像是React.createElement的语法糖。

与引入模板语法对比：react团队认为这分散了开发者的关注点，是一种不佳的方案；

与模板字符串对比：使得内部嵌套过深，过于复杂，也不利于代码提示优化；

对比JSON：同样存在代码提示困难被放弃；

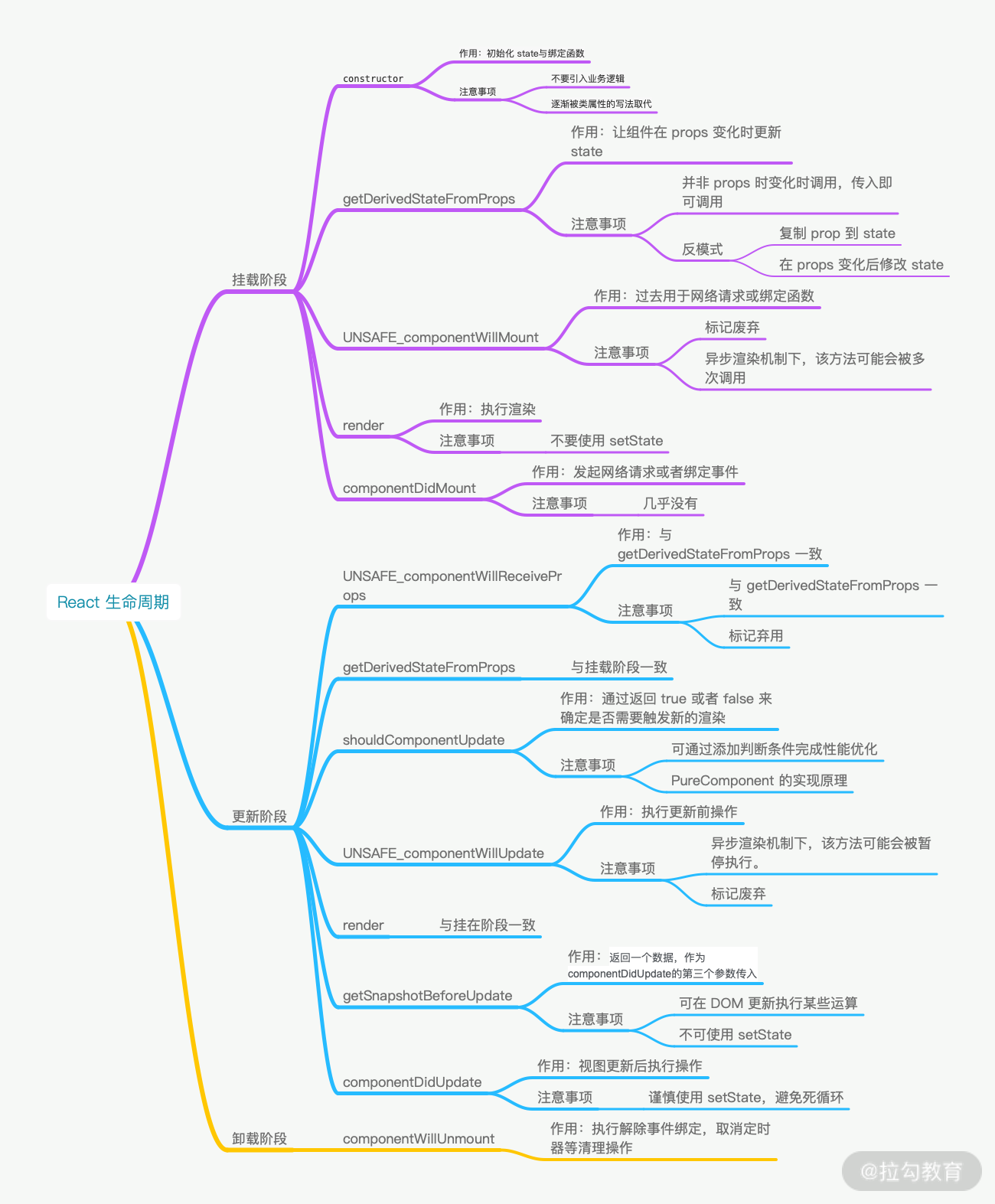
所以：React最终选用了JSX，因为JSX无需过多引入javascript体系以外过多的概念，而是希望通过合理的关注点分离保持组件开发的纯粹性，对编辑器的提示也友好；

Babel 插件如何实现 JSX 到 JS 的编译：

Babel 读取代码并解析，生成 AST，再将 AST 传入插件层进行转换，在转换时就可以将 JSX 的结构转换为 React.createElement 的函数。

生命周期的坑：

1. 在不恰当的时机调用了不合适的代码；
2. 在需要调用时，却忘记了调用。



什么时候出发渲染：

1. 函数组件：函数组件任何情况都会重新渲染，可以使用使 用React.memo来优化。React.memo不是阻断渲染，而是跳 过渲染组件的凑走直接复用最近一次渲染的结果
2. React.Component：如果不实现SCU函数，那么：state发生 变化时触发;当父组件的props传入时，无论props有没有变 化，只要传入就会冲你想你渲染
3. React.PureComponent：默认实现了SCU函数，将props与state进行钱比较后，确认有变更时才会触发重新渲染

类组件和函数组件的区别：

1. 作为组件而言，类组件与函数组件在使用与呈现上没有任何不同，性能上在现代浏览器中也不会有明显差异。
2. 它们在开发时的心智模型上却存在巨大的差异。类组件是基于面向对象编程的，它主打的是继承、生命周期等核心概念；而函数组件内核是函数式编程，主打的是 immutable、没有副作用、引用透明等特点。
3. 之前，在使用场景上，如果存在需要使用生命周期的组件，那么主推类组件；设计模式上，如果需要使用继承，那么主推类组件。
4. 现在由于 React Hooks 的推出，生命周期概念的淡出，函数组件可以完全取代类组件。
5. 其次继承并不是组件最佳的设计模式，官方更推崇“组合优于继承”的设计概念，所以类组件在这方面的优势也在淡出。
6. 性能优化上，类组件主要依靠 shouldComponentUpdate 阻断渲染来提升性能，而函数组件依靠 React.memo 缓存渲染结果来提升性能。
7. 从上手程度而言，类组件更容易上手，从未来趋势上看，由于React Hooks 的推出，函数组件成了社区未来主推的方案。
8. 类组件在未来时间切片与并发模式中，由于生命周期带来的复杂度，并不易于优化。而函数组件本身轻量简单，且在 Hooks 的基础上提供了比原先更细粒度的逻辑组织与复用，更能适应 React 的未来发展。

如何设计React组件：

组件功能分类设计：

展示组件：内部没有状态管理，纯粹的展示表达,专注于组件本身特性

代理组件：代理组件常用于封装常用属性、减少重复代码。很经典的场景就是引入 Antd 的 Button 时，你再自己封一层。如果未来需要替换掉 Antd 或者需要在所有的 Button 上添加一个属性，都会非常方便。

样式组件：细分了处理样式领域，将当前的关注点分离到当前组件内；

布局组件：布局组件的基本设计与样式组件完全一样，布局本身是确定的，不需要根据外部状态的变化去修改内部组件。所以这也是一个可以减少渲染的优化点。通过写死SCU周期函数提高性能；

灵巧组件：更专注于组合组件

容器组件：容器组件几乎没有复用性，拉取数据与组合组件；

高阶组件：React 中复用组件逻辑的高级技术，高阶组件的参数是组件，返回值为新组件的函数，类似高阶函数的概念。作用：抽取公共逻辑、渲染劫持（loading处理）。缺陷：外层无法获取内层的静态函数，可通过 hoist-non-react-statics、refs 属性不能透传：利用React.forwardRef解决

工程化实践：文件夹划分的方式切分代码

如：展示组件：basic，可以使用 Storybook 进行组件的开发管理，提升项目的工程化管理能力。、容器组件：container、高阶组件：hoc等

合成事件：

1. React 给 document 挂上事件监听；
2. DOM 事件触发后冒泡到 document；
3. React 找到对应的组件，造出一个合成事件出来；
4. 并按组件树模拟一遍事件冒泡。
5. 所以这就造成了，在一个页面中，只能有一个版本的 React。如果有多个版本，事件就乱套了。值得一提的是，这个问题在 React 17 中得到了解决，事件委托不再挂在 document 上，而是挂在 DOM 容器上，也就是 ReactDom.Render 所调用的节点上。

setState是同步还是异步的：

1. setState 并非真异步，只是看上去像异步。在源码中，通过 isBatchingUpdates 来判断setState 是先存进 state 队列还是直接更新，如果值为 true 则执行异步操作，为 false 则直接更新。
2. 那么什么情况下 isBatchingUpdates 会为 true 呢？在 React 可以控制的地方，就为 true，比如在 React 生命周期事件和合成事件中，都会走合并操作，延迟更新的策略。
3. 但在 React 无法控制的地方，比如原生事件，具体就是在 addEventListener 、setTimeout、setInterval 等事件中，就只能同步更新。
4. 一般认为，做异步设计是为了性能优化、减少渲染次数，React 团队还补充了两点。
5. 保持内部一致性。如果将 state 改为同步更新，那尽管 state 的更新是同步的，但是 props不是。
6. 启用并发更新，完成异步渲染。

Redux：

概念： redux是一个javascript状态容器，提供可与预测化的状态管理；

用途：提供状态管理，简化高级特性（撤销/重做，实时编辑，时间旅行）

核心理念：

三大原则：单一数据源；纯函数reducer；state是只读的，通过action 修改；

互动方式：actions->reducers->store->view

副作用：redux-thunk、redux-saga

工程化方案： dva、remach

优点：

1. 结果可预测，代码调试简单；
2. 代码结构严格易维护；
3. 模块分离清晰，小函数结构容易便携单元测试；
4. 开发调试容易；
5. 服务端同构容易，传递store即可完成渲染；
6. 社区生态繁荣

虚拟DOM的工作原理：

1. 虚拟 DOM 的工作原理是通过 JS 对象模拟 DOM 的节点。在 Facebook 构建 React 初期时，考虑到要提升代码抽象能力、避免人 为的DOM 操作、降低代码整体风险等因素，所以引入了虚拟 DOM。
2. 虚拟 DOM 在实现上通常是 Plain Object，以 React 为例，在 render 函数中写的 JSX 会在 Babel 插件的作用下，编译为 React.createElement 执行 JSX 中的属性参数。
3. React.createElement 执行后会返回一个 Plain Object，它会描述自己 的 tag 类型、props 属性以及 children 情况等。这些 Plain Object 通 过树形结构组成一颗虚拟 DOM 树。当状态发生变更时，将变更前后 的虚拟 DOM 树进行差异比较，这个过程称为 diff，生成的结果称为 patch。计算之后，会渲染 Patch 完成对真实 DOM 的操作。
4. 虚拟 DOM 的优点主要有三点：改善大规模 DOM 操作的性能、规 避 XSS 风险、能以较低的成本实现跨平台开发。
5. 虚拟 DOM 的缺点在社区中主要有两点。内存占用较高，因为需要 模拟整个网页的真实 DOM。高性能应用场景存在难以优化的情况， 类似像 Google Earth 一类的高性能前端应用在技术选型上往往不会 选择 React。

react的diff算法：

1. diff 算法是指生成更新补丁的方式，主要应用于虚拟 DOM 树变化后， 更新真实 DOM。所以 diff 算法一定存在这样一个过程：触发更新 → 生 成补丁 → 应用补丁。
2. React 的 diff 算法，触发更新的时机主要在 state 变化与 hooks 调用之 后。此时触发虚拟 DOM 树变更遍历，采用了深度优先遍历算法。但传统 的遍历方式，效率较低。为了优化效率，使用了分治的方式。将单一节点 比对转化为了 3 种类型节点的比对，分别是树、组件及元素，以此提升 效率。
3. 树比对：由于网页视图中较少有跨层级节点移动，两株虚拟 DOM 树只 对同一层次的节点进行比较。
4. 组件比对：如果组件是同一类型，则进行树比对，如果不是，则直接放 入到补丁中。
5. 元素比对：主要发生在同层级中，通过标记节点操作生成补丁，节点操 作对应真实的 DOM 剪裁操作。
6. 自 React 16 起，引入了 Fiber 架构。为了使整个更新过程可随时暂停恢复，节点与树分别采用了 FiberNode 与 FiberTree 进行重构。fiberNode 使 用了双链表的结构，可以直接找到兄弟节点与子节点。
7. 整个更新过程由 current 与 workInProgress 两株树双缓冲完成。 workInProgress 更新完成后，再通过修改 current 相关指针指向新节点。
8. Vue 的整体 diff 策略与 React 对齐，虽然缺乏时间切片能力，但这并不 意味着 Vue 的性能更差，因为在 Vue 3 初期引入过，后期因为收益不高 移除掉了。除了高帧率动画，在 Vue 中其他的场景几乎都可以使用防抖 和节流去提高响应性能。

React渲染异常处理：

异常定义：React 渲染异常的时候，在没有做任何拦截的情况下，会出现整个页面白屏的现象。它的成型原因是在渲染层出现了 JavaScript 的错误，导致整个应用崩溃。这种错误通常是在 render 中没有控制好空安全，使值取到了空值。

处理策略：

（1）三个方案：第一个是引入外部函数，比如 Facebook 的 idx 或者 Lodash.get；第二个是引入 Babel 插件，使用 ES 2020 的标准——可选链操作符；第三个是 TypeScript，它在 3.7 版本以后可以直接使用可选链操作符。最后我选择了引入 Babel 插件的方案，因为这个方案外部依赖少，侵入性小，而且团队内没有 TS 的项目。

（2）错误边界函数、组件提示，提高体验；

（3）错误埋点，分析错误原因，进行宣讲培训，降低错误率

1. Webpack

概念：就是一个模块打包机：他做的事情，就是分析你的项目，在内部构建一个依赖图，找到JavaScript模块以及其他一些不能被浏览器直接识别运行的拓展语言，并将其打包为合适的格式以供浏览器使用；

核心组成：

1. Entry：指示webpack应该使用哪个模块，来作为构建其内部依赖图的开始
2. Output：告诉webpack在哪里输出它所创建的bundle，以及如何命名这些文件；
3. Loader：让webpack可以处理除js以及json以外格式的文件，并将他们转化为有效的模块，以供程序使用，以及将他们添加到依赖图中；其实也可以处理js文件；多个loader使用顺序：从右到左执行
4. Plugin：插件机制的目的是为了增强 Webpack 在项目自动化构建方面的能力，可以执行范围更广的任务，比如：打包优化，资源压缩，，注入环境变量；

写plugin：

1. Mode：指定环境；

模块化：

ES6 模块与 CommonJS 模块的差异：

（1）Commonjs模块输出的是一个值的拷贝：即原来模块中的值改变不会影响到已经加载的值，es6模块输出的是一个值的引用：值改变，引用也改变；

（2）Commonjs是运行时加载：CommonJS 模块就是对象；即在输入时是先加载整个模块，生成一个对象，然后再从这个对象上面读取方法，这种加载称为“运行时加载”。es6模块是编译时输出接口：ES6 模块不是对象，而是通过 export 命令显式指定输出的代码，import时采用静态命令的形式。即在import时可以指定加载某个输出值，而不是加载整个模块，这种加载称为“编译时加载”；

（3）CommonJS 加载的是整个模块，即将所有的接口全部加载进来，ES6 可以单独加载其中的某个接口（方法）；

（4）CommonJS this 指向当前模块，ES6 this 指向undefined；

Tree shaking:

机制简述：基于ES6模块的静态引用，tree shaking通过扫描所有的es6 的export，找出被import的内容并添加到最终代码中。webpack的实 现就是把所有的import标记为有使用/无使用两种，在后续的压缩时 进行区别处理掉无使用的代码。

用法：

1. 必须为ES6模块；
2. .babelrc设置babel-preset-es2015的modules为fasle，表示不对 ES6模块进行处理。

注意：

1. 使用 ES6 模块语法编写代码；
2. 工具类函数尽量以单独的形式输出，不要集中成一个对象或者 类；
3. 声明 sideEffects；
4. 自己在重构代码时也要注意副作用；
5. 网络

http：

定义：HyperText Transfer Protocol，超文本传输协议,TCP的应用层协议；

特点：

无连接：每次链接只处理一个请求，请求和应答后就断开链接；

无状态：每次请求都是独立的，不相关的，协议对事务处理没有记忆 功能的，所以有了cookie和session

HTTP/2.0多路复用：每个HTTP请求都有一个序列标识符，这样浏览器 可以并发多个请求，服务器接收到数据后，再根据序列标识符重新排 序成不同的请求报文，而不会导致数据错乱（ 细节参照此文）。同 样，服务端也可以并发返回多个响应给浏览器，浏览器收到后根据序 列标识重新排序并归入各自的请求的响应报文。并且同一个域名下的 所有请求都复用同一个TCP连接，极大增加了服务器处理并发的上 限。

TCP三次握手：

1. 客户端随机生成一个 sequence number，并发送 SYN 报文到服 务端，请求连接
2. 服务端发送 SYN＋ACK，在应答请求的同时，也随机生成一个 sequence id，请求同步
3. 客户端应答，服务端收到应答后双方建立连接。

TCP四次挥手：可以由客户端发起，也可以由服务端发起

1. 发起方发送 FIN 报文，代表断开连接
2. 接收方响应 ACK 报文，并在自己发送完未处理的报文后发送 FIN 报文
3. 发起方接收 ACK 报文后等待接收方的 FIN 报文，收到后发送 ACK 报文，自己进入 TIME\_WAIT 状态，等待 2MSL 后关闭连 接
4. 接收方收到 ACK 报文，关闭连接

请求方法：

HEAD：head方法用于获取报文首部，不返回报文主体；

OPTIONS：options方法用于询问请求URI资源支持的方法；

简单请求：浏览器发送CORS请求（跨域请求时，会区分简单复杂请 求）

1. 请求的方法只能为HEAD、GET、POST
2. 无自定义请求头
3. Content-Type只能是这几种：text/plain multipart/form-data application/x-www-form-urlencoded
4. 先执行, 后判断

复杂请求：

1. PUT, Delete 方法的 ajax 请求；
2. 发送 JSON 格式的 ajax 请求(比如post数据)；
3. 带自定义头的 ajax 请求；
4. 复杂请求在发生请求时, 如果是 CORS 请求，浏览器预先 发送一个 option 请求。浏览器这种行为被称之为预检请求 （注意如果不是跨域请求就不会发生预检请求，比如反向代 理）。

GET和POST的区别：

1. GET和POST没有实质的区别，只是报文格式不同；
2. 一些默认的规范：
3. GET：数据在 URL 中对所有人都是可见的。POST数据不 会显示在 URL 中
4. POST 比 GET 更安全，因为参数不会被保存在浏览器历 史或 web 服务器日志中。
5. GET 方法向 URL 添加数据；URL 的长度是受限制的（URL 的最大长度是 2048 个字符）。
6. GET可被缓存，POST不会；
7. 数据类型POST不限制，可支持二进制；

常见的报文头属性：

Cache-control：

Private：客户端可缓存信息；

Public：客户端和服务端都可缓存；

Max-age：缓存的内容将在XXX秒后失效；

No-cache：可以在本地缓存，可以在代理服务器缓存，但是 这个缓存要服务器验证才可以使用；

no-store：底得禁用缓冲，本地和代理服务器都不缓冲，每 次都从服务器获取；

ETag：

一个代表响应服务端资源（如页面）版本的报文头属性，如 果某个服务端资源发生变化了，这个ETag就会相应发生变 化。配合if-none-match进行判断使用，如无变化，返回304；

Set-cookie：服务端可以设置客户端的Cookie，其原理就是通过 这个响应报文头属性实现的： Set-Cookie: UserID=JohnDoe; Max-Age=3600; Version=1

常见的状态码：

100-199: 信息响应；

200-299:成功响应；

300-299: 重定向：

304 not modified：请求被允许，但文档的内容没有改变

400-499:客户端报错：

401:当前请求需要用户验证；

403:服务器已经理解请求，但是拒绝执行它。与 401 响应 不同的是，身份验证并不能提供任何帮助，而且这个请 求也不应该被重复提交。

404:not found：请求失败，请求所希望得到的资源未被在服 务器上发现；

500-599:服务端报错：

500:服务器遇到了不知道如何处理的情况。

501:此请求方法不被服务器支持且无法被处理。

502:此错误响应表明服务器作为网关需要得到一个处理这 个请求的响应，但是得到一个错误的响应

503: 服务器没有准备好处理请求。 常见原因是服务器因维 护或重载而停机。

缓存：

分类：

强缓存：浏览器在加载资源时，会先根据本地缓存资源的 header 中的信息判断是否命中强缓存，如果命中则直接 使用缓存中的资源不会再向服务器发送请求

关注字段：Expire和Cache-control；

form memory cache：不请求网络资源，资源在内存当 中，一般脚本、字体、图片会存在内存当中；

form disk ceche： 不请求网络资源，在磁盘当中，一 般非脚本会存在内存当中，如css等

协商缓存(对比缓存)：当强缓存没有命中的时候，浏览器会 发送一个请求到服务器，服务器根据请求头中的部分信 息来判断是否命中缓存。如果命中，则返回 304 ，告 诉浏览器资源未更新，可使用本地的缓存。

关注字段：

Last-Modified，If-Modified-Since：两个相同，则 返回304

Etag和If-no-match：Etag 一般是由文件内容 hash 生成的，也就是说它可以保证资源的唯一性，资源 发生改变就会导致 Etag 发生改变。同样地，在浏 览器第一次请求资源时，服务器会返回一个 Etag 标识。当再次请求该资源时， 会通过 If-no-match 字段将 Etag 发送回服务器，然后服务器进行比 较，如果相等，则返回 304 表示未修改。

1. 设计模式

创建型：

工厂模式：创建对象的一种形式，避免重复新建结构相同的对象；

**function** createPeopleFactory(id, name, age) {

**const** obj **=** **new** Object();

obj.id **=** id;

obj.name **=** name;

obj.age **=** age;

**return** obj;}

**const** child **=** createPeopleFactory(1, 'baby', 1);

**const** father **=** createPeopleFactory(2, 'peter', 25);

单例模式：保证一个类仅有一个实例，并提供一个访问它的全局访问点

**const** singleton **=** **function**(fn) {

**let** result **=** **null**;

**return** **function**() {

**return** result **||** (result **=** fn.apply(**this**, arguments));

};};

**const** getScript **=** singleton(**function**() {

**return** document.createElement('script');

});

**const** script1 **=** getScript();

**const** script2 **=** getScript();

console.log(script1 **===** script2); *// true*

结构性：

适配器：分情况做适配

**function** pay(id, price) {

**const** platform **=** window.platform;

**switch** (platform) {

**case** 'wechat'**:**

wx.pay({ id, price });

**break**;

**case** 'alipay'**:**

alipay.pay({ id, price });

**break**;

**case** 'xxx'**:**

xxx.toPay({ goodsId**:** id, price });

**break**;

}}

观察者模式和发布订阅模式区别：

表面上：

1. 观察者模式里，只有两个角色 —— 观察者 + 被观察者；
2. 而发布订阅模式里，却不仅仅只有发布者和订阅者两个角色，还有 一个经常被我们忽略的 —— 经纪人Broker；

更深层：

1. 观察者和被观察者，是松耦合的关系
2. 发布者和订阅者，则完全不存在耦合

使用上：

1. 观察者模式，多用于单个应用内部；
2. 发布订阅模式，则更多的是一种跨应用的模式(cross-application pattern)，比如我们常用的消息中间件