正则表达式：

**什么是正则表达式？**

正则表达式（Regular Expression 或 Regex），是用于定义某种特定搜索模式的字符组合。正则表达式可用于匹配、查找和替换文本中的字符，进行输入数据的验证，查找英文单词的拼写错误等。

**开始**

在 Javascript 中，一个正则表达式以 / 开头和结尾，所以简单至 /hello regexp/ 就是一个正则表达式。

**Flags（标志符或修饰符）**

Flags 写在结束的/之后，可以影响整个正则表达式的匹配行为。常见的 flags 有：

g：全局匹配（global）；正则表达式默认只会返回第一个匹配结果，使用标志符g则可以返回所有匹配

i：忽略大小写（case-insensitive）；在匹配时忽略英文字母的大小写

m：多行匹配（multiline）；将开始和结束字符（^和$）视为在多行上工作，即分别匹配每一行（由 \n 或 \r 分割）的开始和结束，而不只是只匹配整个输入字符串的最开始和最末尾处

**Character Sets（字符集合）**

用于匹配字符集合中的任意一个字符，常见的字符集有：

[xyz]：匹配 "x"或"y"``"z"

[^xyz]：补集，匹配除 "x" "y" "z"的其他字符

[a-z]：匹配从 "a" 到 "z" 的任意字符

[^a-n]：补集，匹配除 "a" 到 "n" 的其他字符

[A-Z]：匹配从 "A" 到 "Z" 的任意字符

[0-9]：匹配从 "0" 到 "9" 的任意数字

**Quantifiers (量词)**

在实际使用中，我们常常需要匹配同一类型的字符多次，比如匹配 11 位的手机号，我们不可能将 [0-9] 写 11 遍，此时我们可以使用 Quantifiers 来实现重复匹配。

{n}：匹配 n 次

{n,m}：匹配 n-m 次

{n,}：匹配 >=n 次

?：匹配 0 || 1 次

\*：匹配 >=0 次，等价于 {0,}

+：匹配 >=1 次，等价于 {1,}

**Metacharacters（元字符）**

在正则表达式中有一些具有特殊含义的字母，被称为元字符，简言之，元字符就是描述字符的字符，它用于对字符表达式的内容、转换及各种操作信息进行描述。

常见的元字符有：

\d：匹配任意数字，等价于 [0-9]

\D：匹配任意非数字字符；\d 的补集

\w：匹配任意基本拉丁字母表中的字母和数字，以及下划线；等价于 [A-Za-z0-9\_]

\W：匹配任意非基本拉丁字母表中的字母和数字，以及下划线；\w 的补集

\s：匹配一个空白符，包括空格、制表符、换页符、换行符和其他 Unicode 空格

\S：匹配一个非空白符；\s的补集

\b：匹配一个零宽单词边界，如一个字母与一个空格之间；例如，/\bno/ 匹配 "at noon" 中的 "no"，/ly\b/ 匹配 "possibly yesterday." 中的 "ly"

\B：匹配一个零宽非单词边界，如两个字母之间或两个空格之间；例如，/\Bon/ 匹配 "at noon" 中的 "on"，/ye\B/ 匹配 "possibly yesterday." 中的 "ye"

\t：匹配一个水平制表符（tab)

\n：匹配一个换行符（newline)

\r：匹配一个回车符（carriage return)

**Special Characters (特殊字符)**

正则中存在一些特殊字符，它们不会按照字面意思进行匹配，而有特殊的意义，比如前文讲过用于量词的?、\*、+。其他常见的特殊字符有：

\：转义字符，可以将普通字符转成特殊字符。比如 \w；也可以将特殊字符转成字面意思，比如 \+ 匹配 "+"

.：匹配任意单个字符，但是换行符除外：\n, \r, \u2028 或 \u2029；在字符集中（[.]），无特殊含义，即表示 '.' 的字面意思

|：替换字符（alternate character），匹配 | 前或后的表达式。比如需要同时匹配 "bear" 和 "pear"，可以使用 /(b|p)ear/ 或者 /bear|pear/；但是不能用 /b|pear/，该表达式只能匹配 "b" 和 "pear"

^：匹配输入的开始。比如，/^A/ 不匹配 "an Apple" 中的 "A"，但匹配 "An apple" 中的 "A"

$：匹配输入的结尾。比如，/t$/ 不匹配 "eater" 中的 "t"，但匹配 "eat" 中的 "t"。^ 和 $ 在表单验证时常需要使用，因为需要验证从开始到结尾的一个完整输入，而不是匹配输入中的某一段

**Groups（分组）**

(xyz)：捕获分组（Capturing Group），匹配并捕获匹配项；例如，/(foo)/ 匹配且捕获 "foo bar." 中的 "foo"。被匹配的子字符串可以在结果数组的元素 [1], ..., [n] 中找到，或在被定义的 RegExp 对象的属性 $1, ..., $9 中找到

(?:xyz)：非捕获分组（Non-capturing Group），匹配但不会捕获匹配项；匹配项不能再次被访问到

\n：n 是一个正整数，表示反向引用（back reference），指向正则表达式中第 n 个括号（从左开始数）中匹配的子字符串；例如，/apple(,)\sorange\1/ 匹配 "apple, orange, cherry, peach." 中的 "apple,orange,"

**Assertion（断言）**

x(?=y)：仅匹配被y跟随的x；例如，/bruce(?=wayne)/，如果"bruce"后面跟着wayne，则匹配之。/bruce(?=wayne|banner)/ ，如果"bruce"后面跟着"wayne"或者banner，则匹配之。但是，"wayne" 和 "banner" 都不会在匹配结果中出现

x(?!y)：仅匹配不被y跟随的x；例如，/\d+(?!\.)/ 只会匹配不被 "." 跟随的数字。

**热部署：**直接重新加载整个应用（生产环境），清空内存重新打包，重新解压war包

**热加载：**在运行时重新加载class（开发环境），基于字节码的更改，不释放内存开发可用,上线不可用，热加载不重启tomcat,不重新打包

**懒加载：**延迟加载，

**热更新：**热更新就是当你在开发环境修改代码后，不用刷新整个页面即可看到修改后的效果

任何执行和处理 DOM 元素的 JS 代码都应该在创建 DOM 元素之后执行。

是的，对于相同的逻辑错误，不同的浏览器可能具有不同的错误消息。

许多函数只接受其输入值的特定范围的数字。 例如：

toExponential(digits) 和 toFixed(digits) 接受 0 到 100

toPrecision(digits) 接受 1 到 100

使用 new Date() 创建日期:

有四种可能的语法：

1)使用字符串日期值

2)使用一系列参数

3)使用时间戳

4)不带参数

最好不要使用字符串日期值的方法创建日期

最好使用一系列参数方式创建日期

记住月份是从0开始的

**格式化日期**

const date = new Date(2019, 0, 23, 17, 23, 42)

toString：格式化成 "Wed Jan 23 2019 17:23:42 GMT+0800 (中国标准时间)"

toDateString: 格式化成 "Wed Jan 23 2019"

toLocaleString：格式化成 "2019/1/23 下午5:23:42"

toLocaleDateString：格式化成 "2019/1/23"

toGMTString：格式化成 "Wed, 23 Jan 2019 09:23:42 GMT"

toUTCString：格式化成 "Wed, 23 Jan 2019 09:23:42 GMT"

toISOString ：格式化成 "2019-01-23T09:23:42.000Z"

**日期的比较**

比较日期的前后，可以直接使用>, <, >= 和 <= 时行比较。

比较两个日期是否一样，就比较麻烦，不能直接用 == 或 ===。可以getTime获取它们的时间戳，用时间戳进行比较。

如果只想检查两个日期是否在同一天，可以比较他们的getFullYear，getMonth和getDate值。

**ES6之Array.from()方法**

1. 类数组对象：所谓类数组对象，最基本的要求就是具有length属性的对象。
2. Array.from()方法就是将一个类数组对象或者可遍历对象转换成一个真正的数组。Array.from有三个参数，Array.from(arrayLike[, mapFn[, thisArg]])，

arrayLike：想要转换成数组的伪数组对象或可迭代对象；

mapFn：如果指定了该参数，新数组中的每个元素会执行该回调函数；

thisArg：可选参数，执行回调函数 mapFn 时 this 对象。该方法的返回值是一个新的数组实例（真正的数组）。

<https://www.jianshu.com/p/f4554f74de95>

JSON.stringify() 方法用于将 JavaScript 值转换为 JSON 字符串。

JSON.parse() 方法用于将一个 JSON 字符串转换为对象。

var potatos = [{ id: '1001', weight: 50 },

{ id: '1002', weight: 80 },

{ id: '1003', weight: 120 },

{ id: '1004', weight: 40 },

{ id: '1005', weight: 110 },

{ id: '1006', weight: 60 }]

**forEach方法 批量操作**

potatos.forEach(potato => potato.weight += 20 )

**map最适合做的事是映射，生成原始数据的特征信息的map**

w = potatos.map(potato => { return potato.weight += 20 })

**filter是滤波的意思 从名字上看，就知道筛选过滤这样的活是filter来干的**

var bigOnes = potatos.filter(potato => { return potato.weight > 100 })

返回一个新的对象数组，并不会改变原数组

我们的**some**小伙计，去土豆存放的仓库进行寻找，**只要找到一个符合条件的，就回来报告true 所以并不会全部遍历**，不做多余的活（性能优良）

var hasbig = potatos.some(potato => { return potato.weight > 100 })

**every对每一个元素执行一个callback，直到它找到一个使 callback 返回 false的元素（没那么大的土豆），就返回false，直到遍历完成也没有返回false的话，就返回true**

var allbig = potatos.every(potato => { return potato.weight > 100 })

find和some很类似，都是寻找符合条件的，有一个就可以 不过some进去搜罗了一圈回来报了个“有”（true），而**find则把那个土豆抱了出来（返回第一个符合条件的对象）**

var big = potatos.find(potato => { return potato.weight > 100 })

//{ id: '1003', weight: 120 }

**当需要知道所需元素的索引，就可以用findIndex**

**findIndex返回第一个符合条件的索引号**

**递归累加 reduce**

reduce()方法接收一个回调函数作为第一个参数，回调函数又接受四个参数，分别是；

1、previousValue =>初始值或上一次回调函数叠加的值；

2、currentValue => 本次回调（循环）将要执行的值；

3、index=>“currentValue”的索引值；

4、arr => 数组本身；

reduce()方法返回的是最后一次调用回调函数的返回值；

设计模式的定义是：在面向对象软件设计过程中针对特定问题的简洁而优雅的解决方案。通俗一点说，设计模式是在某种场合下对某个问题的一种解决方案。如果再通俗一点说，设计模式就是给面向对象软件开发中的一些好的设计取个名字。

**观察者模式（Observer Pattern）**

观察者模式定义了对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都将得到通知，并自动更新。观察者模式属于行为型模式，行为型模式关注的是对象之间的通讯，观察者模式就是观察者和被观察者之间的通讯。

观察者模式有一个别名叫“发布-订阅模式”，或者说是“订阅-发布模式”，订阅者和订阅目标是联系在一起的，当订阅目标发生改变时，逐个通知订阅者。我们可以用报纸期刊的订阅来形象的说明，当你订阅了一份报纸，每天都会有一份最新的报纸送到你手上，有多少人订阅报纸，报社就会发多少份报纸，报社和订报纸的客户就是上面文章开头所说的“一对多”的依赖关系。

**发布订阅模式（Pub-Sub Pattern）**

其实24种基本的设计模式中并没有发布订阅模式，上面也说了，他只是观察者模式的一个别称。

但是经过时间的沉淀，似乎他已经强大了起来，已经独立于观察者模式，成为另外一种不同的设计模式。

在现在的发布订阅模式中，称为发布者的消息发送者不会将消息直接发送给订阅者，这意味着发布者和订阅者不知道彼此的存在。在发布者和订阅者之间存在第三个组件，称为消息代理或调度中心或中间件，它维持着发布者和订阅者之间的联系，过滤所有发布者传入的消息并相应地分发它们给订阅者。

举一个例子，你在微博上关注了A，同时其他很多人也关注了A，那么当A发布动态的时候，微博就会为你们推送这条动态。A就是发布者，你是订阅者，微博就是调度中心，你和A是没有直接的消息往来的，全是通过微博来协调的（你的关注，A的发布动态）。

**观察者模式和发布订阅模式有什么区别？**



观察者模式：观察者（Observer）直接订阅（Subscribe）主题（Subject），而当主题被激活的时候，会触发（Fire Event）观察者里的事件。

发布订阅模式：订阅者（Subscriber）把自己想订阅的事件注册（Subscribe）到调度中心（Topic），当发布者（Publisher）发布该事件（Publish topic）到调度中心，也就是该事件触发时，由调度中心统一调度（Fire Event）订阅者注册到调度中心的处理代码。

观察者模式和发布订阅模式最大的区别就是发布订阅模式有个事件调度中心。

观察者模式由具体目标调度，每个被订阅的目标里面都需要有对观察者的处理，这种处理方式比较直接粗暴，但是会造成代码的冗余。

而发布订阅模式中统一由调度中心进行处理，订阅者和发布者互不干扰，消除了发布者和订阅者之间的依赖。这样一方面实现了解耦，还有就是可以实现更细粒度的一些控制。比如发布者发布了很多消息，但是不想所有的订阅者都接收到，就可以在调度中心做一些处理，类似于权限控制之类的。还可以做一些节流操作。

**观察者模式是不是发布订阅模式**

如果以结构来分辨模式，发布订阅模式相比观察者模式多了一个中间件订阅器，所以发布订阅模式是不同于观察者模式的；如果以意图来分辨模式，他们都是实现了对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都将得到通知，并自动更新，那么他们就是同一种模式，发布订阅模式是在观察者模式的基础上做的优化升级。

不过，不管他们是不是同一个设计模式，他们的实现方式确实有差别，我们在使用的时候应该根据场景来判断选择哪个。

BFC(Block Formatting Context) 块级格式化上下文

**1. 什么是数据结构？**

数据结构是在计算机中组织和存储数据的一种特殊方式，使得数据可以高效地被访问和修改。更确切地说，数据结构是数据值的集合，表示数据之间的关系，也包括了作用在数据上的函数或操作。

**1.1 为什么我们需要数据结构？**

数据是计算机科学当中最关键的实体，而数据结构则可以将数据以某种组织形式存储，因此，数据结构的价值不言而喻。

无论你以何种方式解决何种问题，你都需要处理数据——无论是涉及员工薪水、股票价格、购物清单，还是只是简单的电话簿问题。

数据需要根据不同的场景，按照特定的格式进行存储。有很多数据结构能够满足以不同格式存储数据的需求。

**数组是最简单的数据结构**

**2.1 for... in为何这么慢？**

for... in语法令人难以置信的缓慢。在测试中就已经比正常情况下慢近9倍的循环。这是因为for ... in语法是第一个能够迭代对象键的JavaScript语句。循环对象键（{}）与在数组（[]）上进行循环不同，因为引擎会执行一些额外的工作来跟踪已经迭代的属性。

**3. 堆栈：Stack**

堆栈是元素的集合，可以在顶部添加项目，我们有几个实际的堆栈示例：

浏览器历史记录

撤消操作

递归以及其它。

三句话解释堆栈：

两个原则操作：push和pop。Push 将元素添加到数组的顶部，而Pop将它们从同一位置删除。

遵循"Last In，First Out"，即：LIFO，后进先出。

没了。

4. 队列：Queue

在计算机科学中，一个队列(queue)是一种特殊类型的抽象数据类型或集合。集合中的实体按顺序保存。

而在前端开发中，最著名的队列使用当属浏览器/NodeJs中 关于宏任务与微任务，任务队列的知识。这里就不再赘述了。

在后端领域，用得最广泛的就是消息队列：Message queue：如RabbitMQ、ActiveMQ等。

以编程思想而言，Queue可以用两句话描述：

只是具有两个主要操作的数组：unshift和pop。

遵循"Fist In，first out"即：FIFO，先进先出。

**彻底搞懂浏览器Event-loop(事件循环)**

JavaScript的运行机制:

（1）所有同步任务都在主线程上执行，形成一个执行栈（execution context stack）。

（2）主线程之外，还存在"任务队列"(task queue)。只要异步任务有了运行结果，就在"任务队列"之中放置一个事件。

（3）一旦"执行栈"中的所有同步任务执行完毕，系统就会读取"任务队列"，看看里面有哪些事件。那些对应的异步任务，于是结束等待状态，进入执行栈，开始执行。

（4）主线程不断重复上面的第三步

概括即是: 调用栈中的同步任务都执行完毕，栈内被清空了，就代表主线程空闲了，这个时候就会去任务队列中按照顺序读取一个任务放入到栈中执行。每次栈内被清空，都会去读取任务队列有没有任务，有就读取执行，一直循环读取-执行的操作

一个事件循环中有一个或者是多个任务队列

JavaScript中有两种异步任务:

宏任务: script（整体代码）, setTimeout, setInterval, setImmediate, I/O, UI rendering

微任务（microtasks）: process.nextTick（Nodejs）, Promises, Object.observe, MutationObserver;

**2. 事件循环(event-loop)是什么？**

主线程从"任务队列"中读取执行事件，这个过程是循环不断的，这个机制被称为事件循环。此机制具体如下:主线程会不断从任务队列中按顺序取任务执行，每执行完一个任务都会检查microtask队列是否为空（执行完一个任务的具体标志是函数执行栈为空），如果不为空则会一次性执行完所有microtask。然后再进入下一个循环去任务队列中取下一个任务执行。

详细说明:

1)选择当前要执行的宏任务队列，选择一个最先进入任务队列的宏任务，如果没有宏任务可以选择，则会跳转至microtask的执行步骤。

2)将事件循环的当前运行宏任务设置为已选择的宏任务。

3)运行宏任务。

4)将事件循环的当前运行任务设置为null。

5)将运行完的宏任务从宏任务队列中移除。

6)microtasks步骤：进入microtask检查点。

7)更新界面渲染。

8)返回第一步。

执行进入microtask检查的的具体步骤如下:

1)设置进入microtask检查点的标志为true。

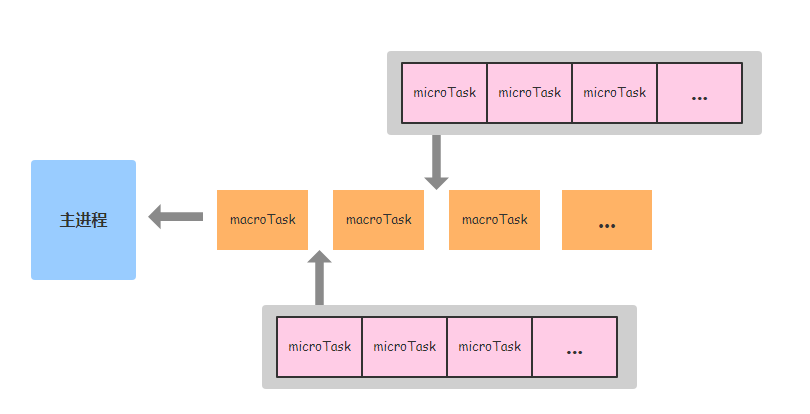
2)当事件循环的微任务队列不为空时：选择一个最先进入microtask队列的microtask；设置事件循环的当前运行任务为已选择的microtask；运行microtask；设置事件循环的当前运行任务为null；将运行结束的microtask从microtask队列中移除。

3)对于相应事件循环的每个环境设置对象（environment settings object）,通知它们哪些promise为rejected。

4)清理indexedDB的事务。

5)设置进入microtask检查点的标志为false。

**需要注意的是:当前执行栈执行完毕时会立刻先处理所有微任务队列中的事件, 然后再去宏任务队列中取出一个事件。同一次事件循环中, 微任务永远在宏任务之前执行。**



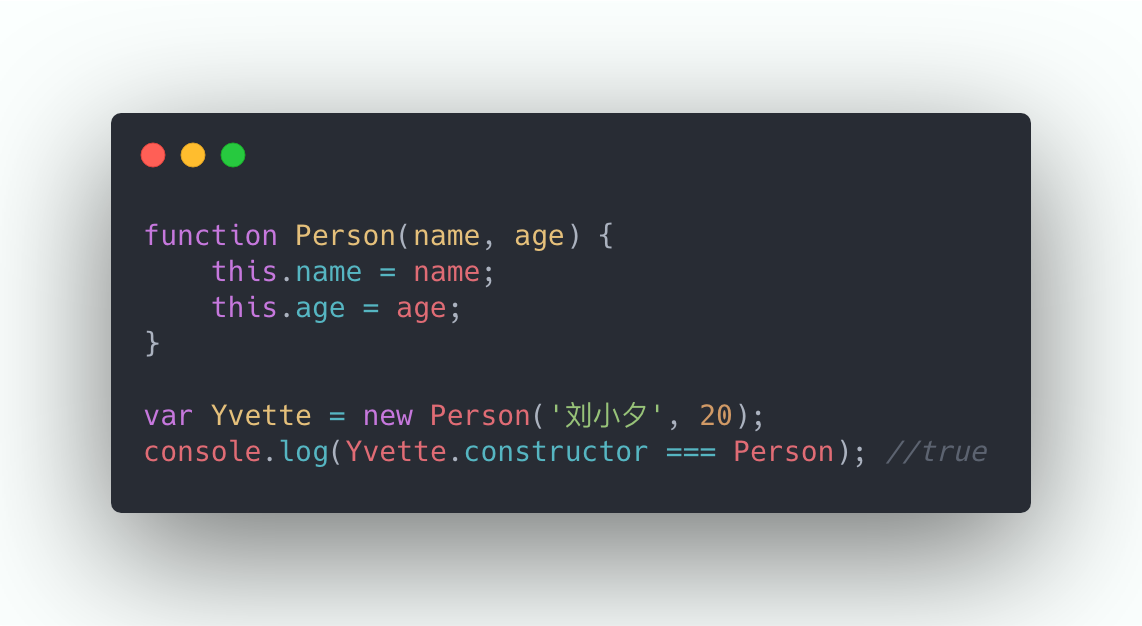
**4. 为什么会需要event-loop?**

因为 JavaScript 是单线程的。单线程就意味着，所有任务需要排队，前一个任务结束，才会执行后一个任务。如果前一个任务耗时很长，后一个任务就不得不一直等着。为了协调事件（event），用户交互（user interaction），脚本（script），渲染（rendering），网络（networking）等，用户代理（user agent）必须使用事件循环（event loops）。

**构造函数**

构造函数和普通函数的区别仅在于调用它们的方式不同，任何函数，只要通过 new 操作符来调用，那它就可以作为构造函数；任何函数，如果不通过 new 操作符来调用，那么它就是一个普通函数。

**实例拥有 constructor(构造函数) 属性，该属性返回创建实例对象的构造函数。**



有一点需要说明的是，除了基本数据类型的 constructor 外( null 和 undefined 无 constructor 属性)，constructor 属性是可以被重写的。因此检测对象类型时，instanceof 操作符比 constructor 更可靠一些。



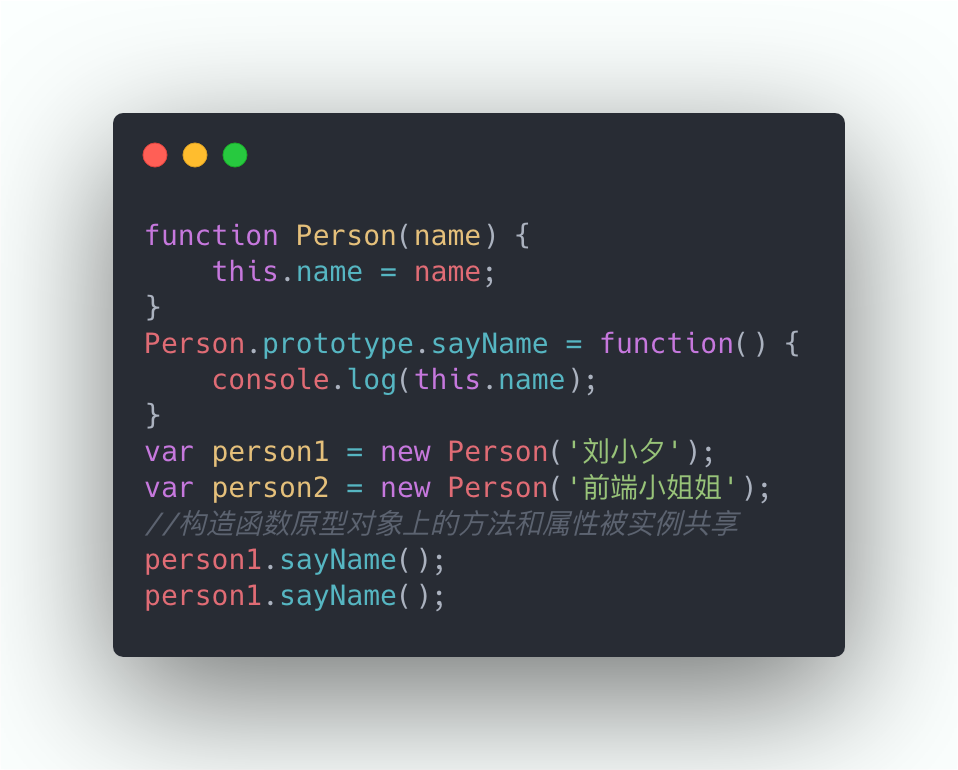
**原型**

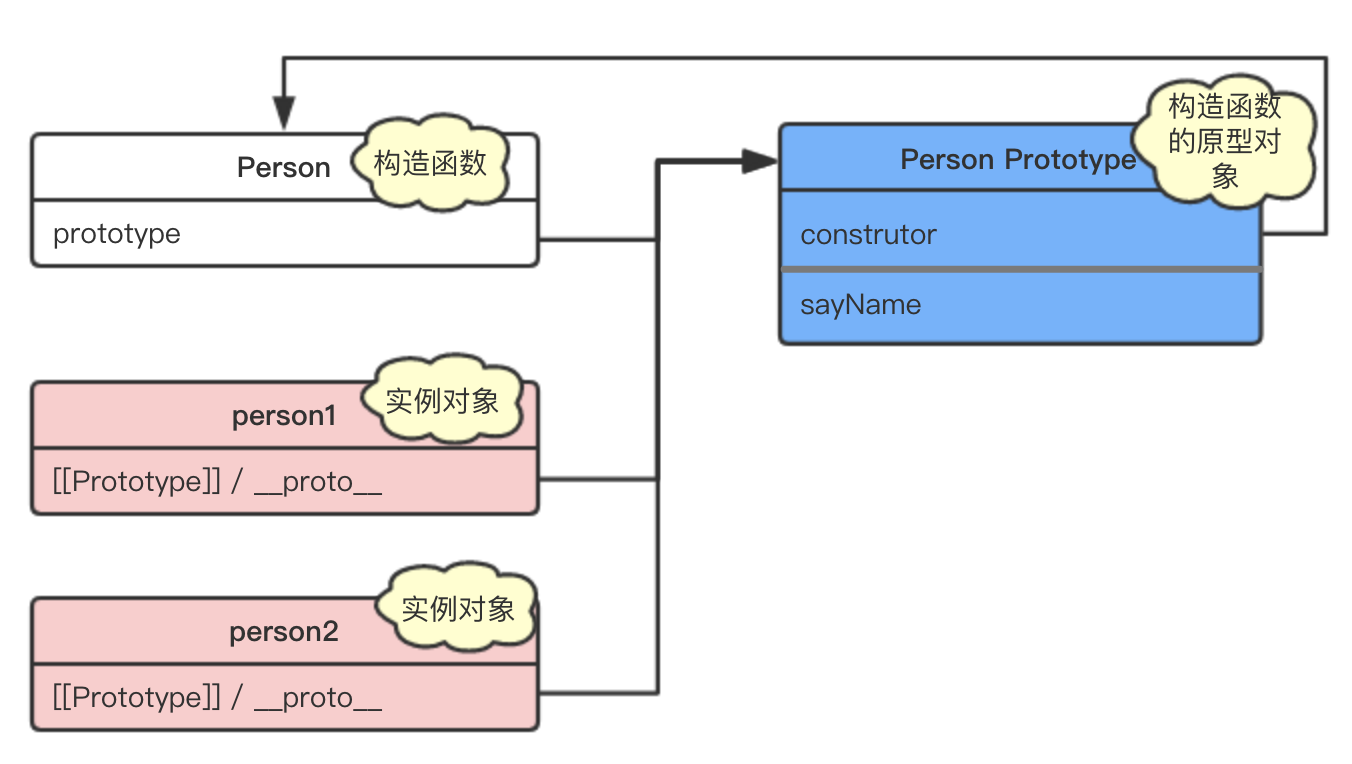
我们创建的每个函数都有 prototype 属性，这个属性指向函数的原型对象。原型对象的用途是包含可以由特定类型的所有实例共享的属性和方法。

在默认情况下，所有原型对象都会自动获得一个 constructor 属性，这个属性包含一个指向 prototype 属性所在函数的指针。

当调用构造函数创建一个新实例后，该实例的内部将包含一个指针，指向构造函数的原型对象(可以通过实例的 \_\_proto\_\_ 来访问构造函数的原型对象)。

当调用构造函数创建一个新实例后，该实例的内部将包含一个指针，指向构造函数的原型对象。





实例.\_\_proto\_\_ === 构造函数.prototype

console.log(Object.prototype.\_\_proto\_\_ === null) //true

console.log(Object.\_\_proto\_\_ === Function.prototype) //true

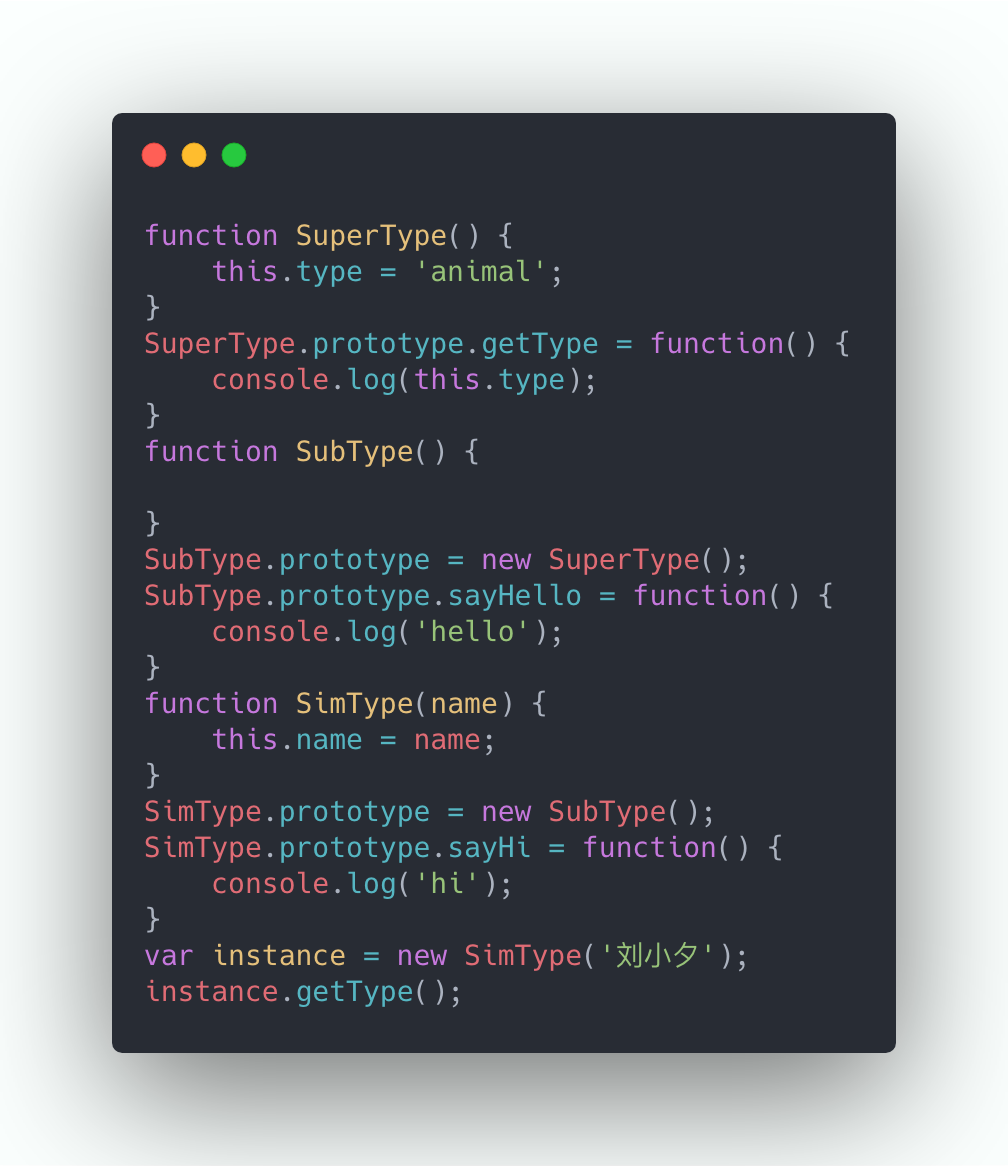
console.log(Function.prototype.\_\_proto\_\_ === Object.prototype) //true

**原型链**

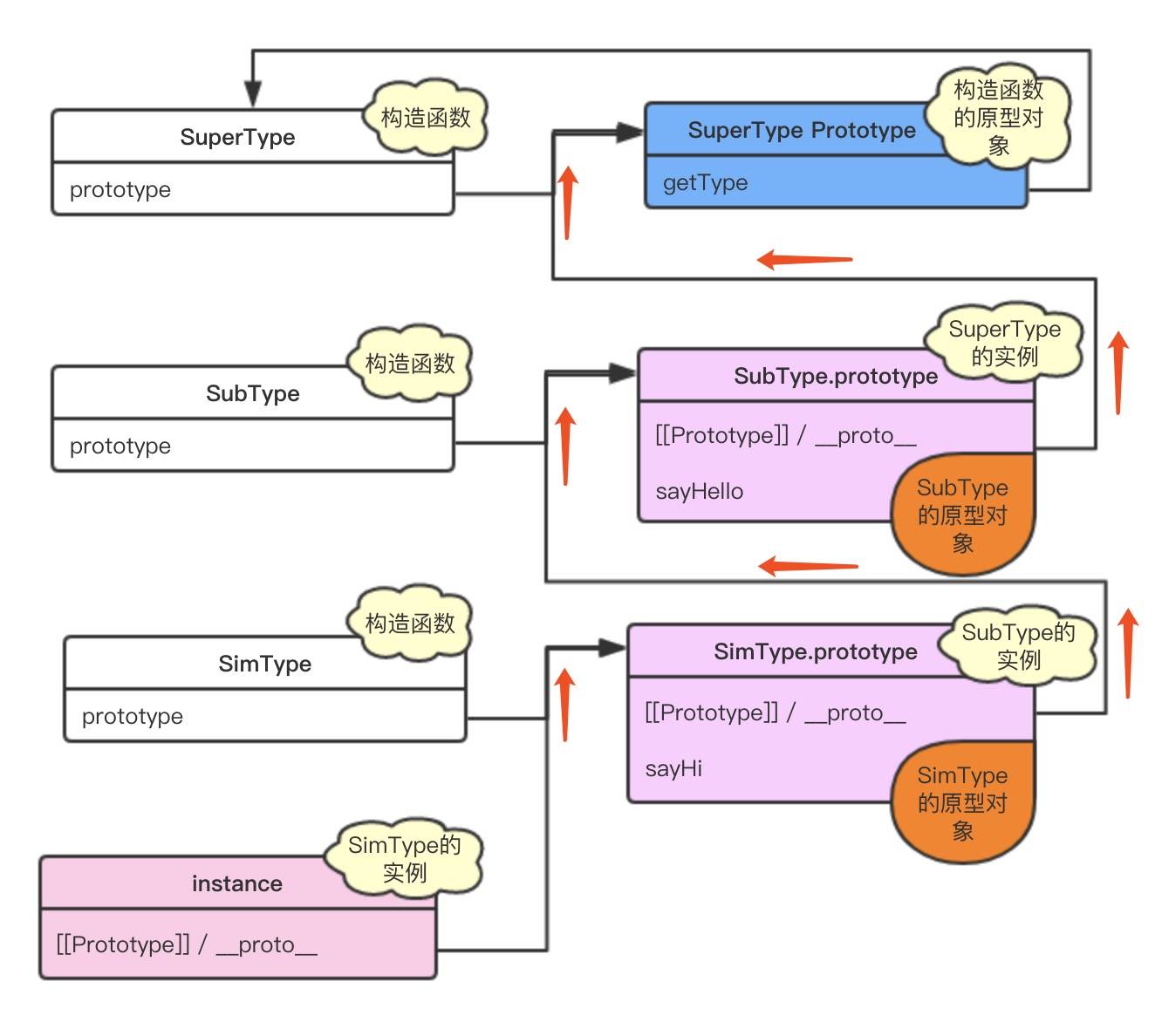
简单回顾一下构造函数、原型和实例的关系：

每个构造函数都有一个原型对象，原型对象都包含一个指向构造函数的指针，而实例都包含一个可以执行原型对象的内部指针(可以通过 \_\_proto 访问)。

假如我们让原型对象等于另一个类型的实例，那么此时原型对象包含一个指向另一个原型的指针，相应地，另一个原型中也包含着一个指向另一个构造函数的指针。加入另一个原型又是另一个类型的实例，那么上述关系仍然成立，如此层层递进，就构成了实例与原型的链条，这就是原型链的基本概念。



一图胜万言:



调用 instance.getType() 会调用以下的搜索步骤:

搜索 instance 实例

搜索 SimType.prototype

搜索 SubType.prototype

搜索 SuperType.prototype，找到了 getType 方法

在找不到属性或方法的情况下，搜索过程总是要一环一环地前行到原型链的末端才会停下来。

所有引用类型都继承了 Object，这个继承也是通过原型链实现的。如果在 SuperType.prototype 还没有找到 getType，就会到 Object.prototype中找(图中少画了一环)。

**原型链继承**

原型链继承的基本思想是利用原型让一个引用类型继承另一个引用类型的属性和方法。

如 SubType.prototype = new SuperType();



可以看出 colors 属性会被所有的实例共享(instance1、instance2、...)。

缺点：

通过原型来实现继承时，原型会变成另一个类型的实例，原先的实例属性变成了现在的原型属性，该原型的引用类型属性会被所有的实例共享。

在创建子类型的实例时，没有办法在不影响所有对象实例的情况下给超类型的构造函数中传递参数。

**借用构造函数**

借用构造函数的技术，其基本思想为:在子类型的构造函数中调用超类型构造函数。



优点:1）可以向超类传递参数

2）解决了原型中包含引用类型值被所有实例共享的问题

缺点:1）方法都在构造函数中定义，函数复用无从谈起，另外超类型原型中定义的方法对于子类型而言都是不可见的。

**组合继承**

组合继承指的是将原型链和借用构造函数技术组合到一块，从而发挥二者之长的一种继承模式。基本思路：

使用原型链实现对原型属性和方法的继承，通过借用构造函数来实现对实例属性的继承，既通过在原型上定义方法来实现了函数复用，又保证了每个实例都有自己的属性。



缺点:无论什么情况下，都会调用两次超类型构造函数：一次是在创建子类型原型的时候，另一次是在子类型构造函数内部。



优点:

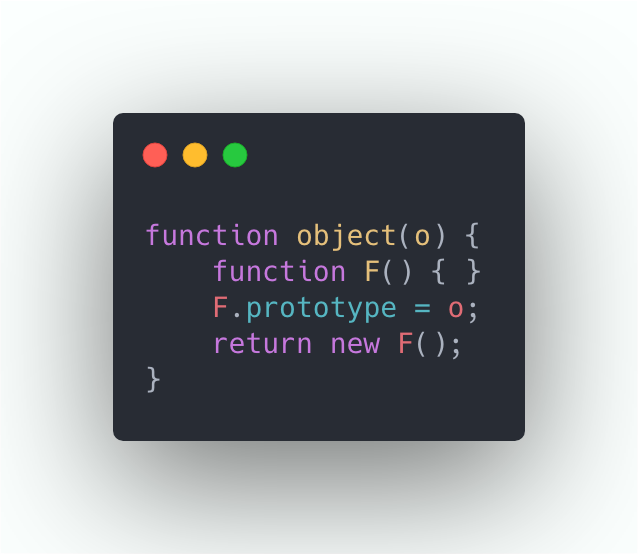
1）可以向超类传递参数

2）每个实例都有自己的属性

3）实现了函数复用

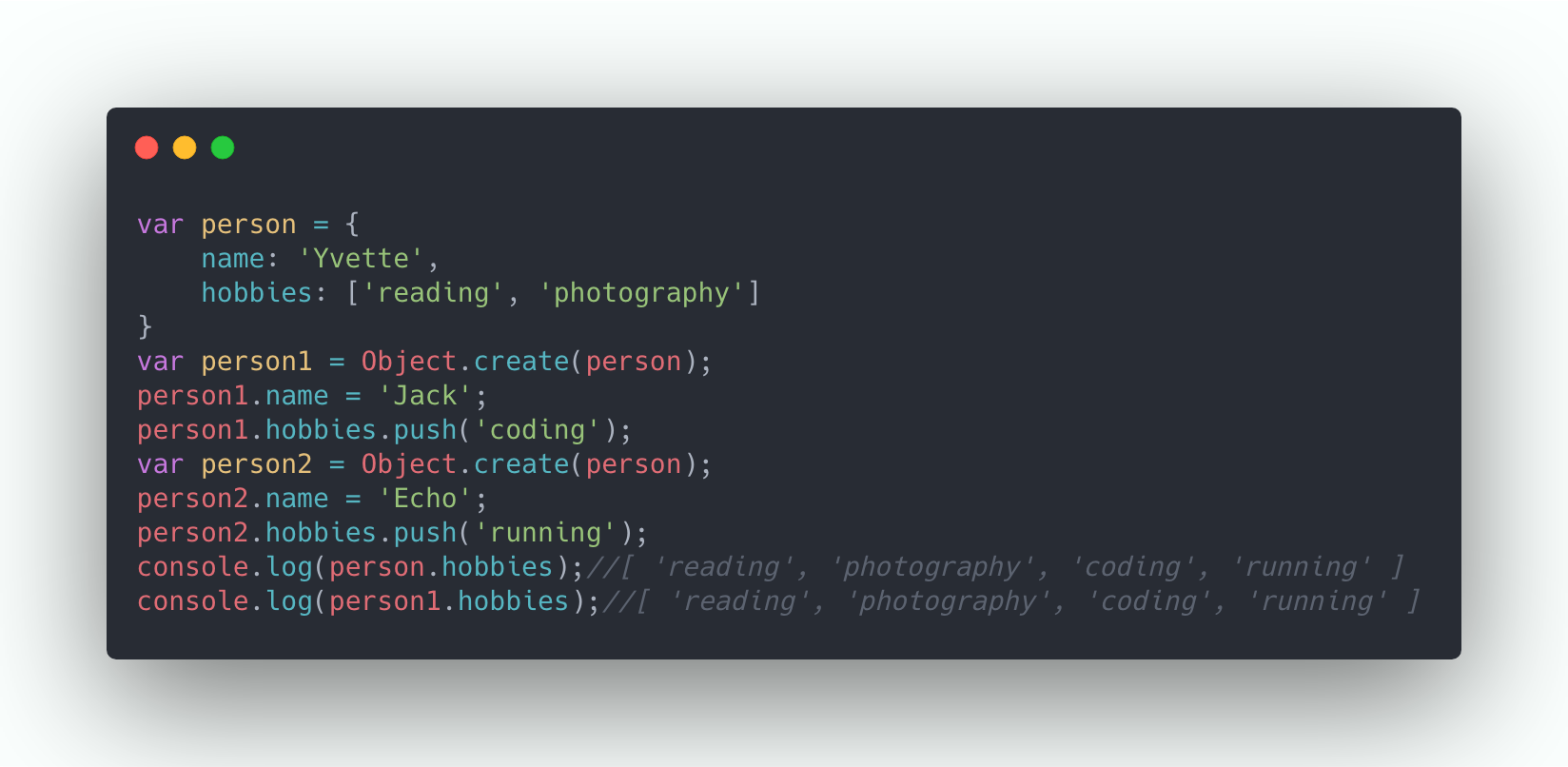
**原型式继承**

原型继承的基本思想：借助原型可以基于已有的对象创建新对象，同时还不必因此创建自定义类型。



在 object() 函数内部，先穿甲一个临时性的构造函数，然后将传入的对象作为这个构造函数的原型，最后返回了这个临时类型的一个新实例，从本质上讲，object() 对传入的对象执行了一次浅拷贝。

ECMAScript5通过新增 Object.create()方法规范了原型式继承。这个方法接收两个参数：一个用作新对象原型的对象和（可选的）一个为新对象定义额外属性的对象(可以覆盖原型对象上的同名属性)，在传入一个参数的情况下，Object.create() 和 object() 方法的行为相同。



在没有必要创建构造函数，仅让一个对象与另一个对象保持相似的情况下，原型式继承是可以胜任的。

缺点:同原型链实现继承一样，包含引用类型值的属性会被所有实例共享。

**寄生式继承**

寄生式继承是与原型式继承紧密相关的一种思路。寄生式继承的思路与寄生构造函数和工厂模式类似，即创建一个仅用于封装继承过程的函数，该函数在内部已某种方式来增强对象，最后再像真地是它做了所有工作一样返回对象。



基于 person 返回了一个新对象 -—— person2，新对象不仅具有 person 的所有属性和方法，而且还有自己的 sayHi() 方法。在考虑对象而不是自定义类型和构造函数的情况下，寄生式继承也是一种有用的模式。

缺点：

1）使用寄生式继承来为对象添加函数，会由于不能做到函数复用而效率低下。

2）同原型链实现继承一样，包含引用类型值的属性会被所有实例共享。

**寄生组合式继承**

所谓寄生组合式继承，即通过借用构造函数来继承属性，通过原型链的混成形式来继承方法，基本思路：

不必为了指定子类型的原型而调用超类型的构造函数，我们需要的仅是超类型原型的一个副本，本质上就是使用寄生式继承来继承超类型的原型，然后再将结果指定给子类型的原型。寄生组合式继承的基本模式如下所示：



第一步：创建超类型原型的一个副本

第二步：为创建的副本添加 constructor 属性

第三步：将新创建的对象赋值给子类型的原型

至此，我们就可以通过调用 inheritPrototype 来替换为子类型原型赋值的语句



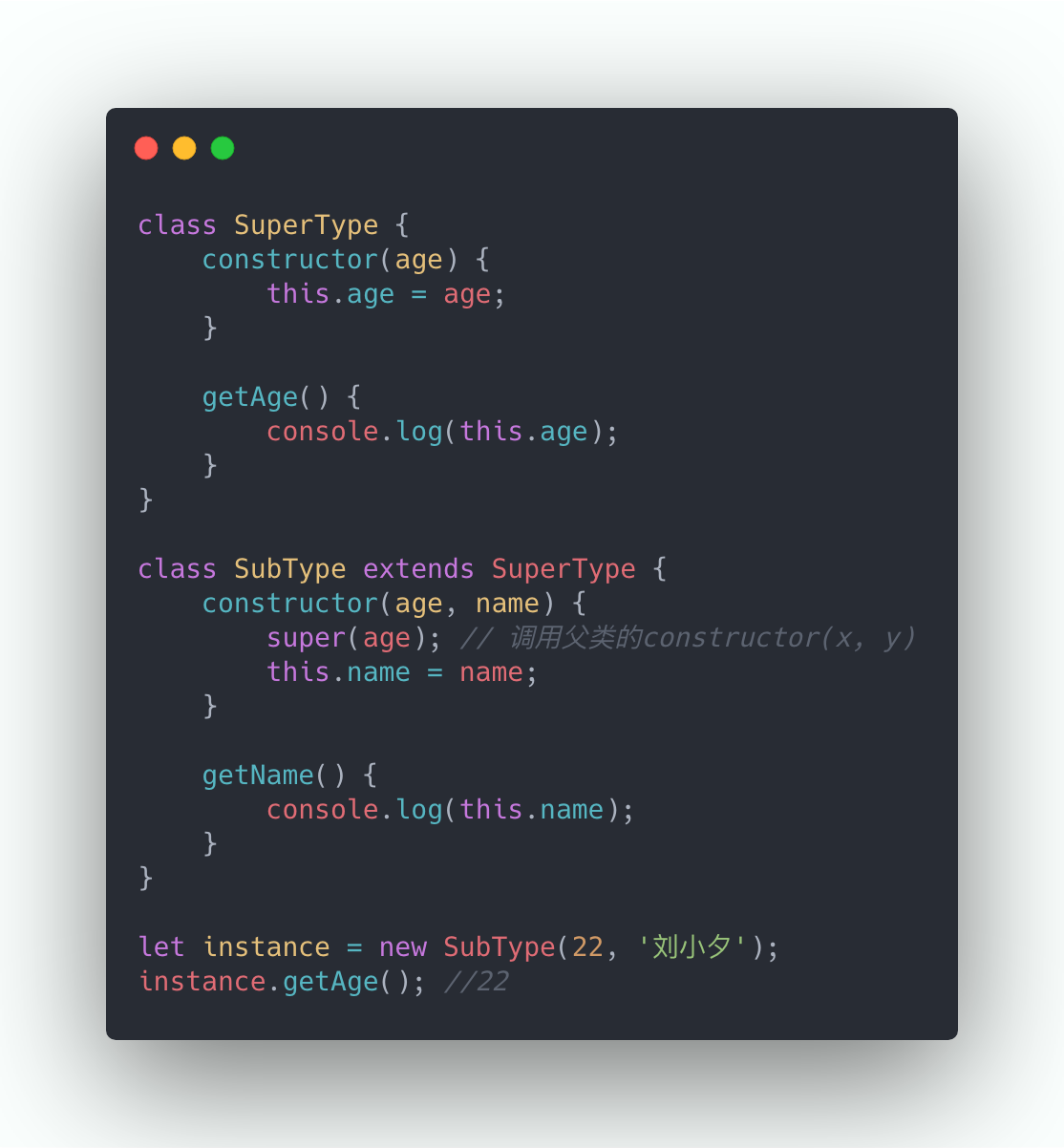
优点:

只调用了一次超类构造函数，效率更高。避免在SuberType.prototype上面创建不必要的、多余的属性，与其同时，原型链还能保持不变。

因此寄生组合继承是引用类型最理性的继承范式。

**ES6 继承**

Class 可以通过extends关键字实现继承，如:



对于ES6的 class 需要做以下几点说明：

1）类的数据类型就是函数，类本身就指向构造函数。

console.log(typeof SuperType);//function

console.log(SuperType === SuperType.prototype.constructor); //true

2）类的内部所有定义的方法，都是不可枚举的。(ES5原型上的方法默认是可枚举的)

Object.keys(SuperType.prototype);

3）constructor 方法是类的默认方法，通过 new 命令生成对象实例时，自动调用该方法。一个类必须有constructor 方法，如果没有显式定义，一个空的 constructor 方法会被默认添加。

4）Class 不能像构造函数那样直接调用，会抛出错误。

使用 extends 关键字实现继承，有一点需要特别说明：

子类必须在 constructor 中调用 super 方法，否则新建实例时会报错。如果没有子类没有定义 constructor 方法，那么这个方法会被默认添加。在子类的构造函数中，只有调用 super 之后，才能使用 this关键字，否则报错。这是因为子类实例的构建，基于父类实例，只有super方法才能调用父类实例。

**webpack 打包原理**

初始化：启动构建，读取与合并配置参数，加载 Plugin，实例化 Compiler。

编译：从 Entry 发出，针对每个 Module 串行调用对应的 Loader 去翻译文件内容，再找到该 Module 依赖的 Module，递归地进行编译处理。

输出：对编译后的 Module 组合成 Chunk，把 Chunk 转换成文件，输出到文件系统。

**常见的内存泄露**

全局变量，未清除的定时器，闭包，以及 dom 的引用等。

箭头函数：

传参：不需要参数或多个参数时，使用括号（）包起来。当有一个参数时，可以省略括号。

**箭头函数与传统的JavaScript函数主要区别在于以下几点：**

1、对 this 的关联。函数内置 this 的值，取决于箭头函数在哪儿定义，而非箭头函数执行的上下文环境。任何方法都改变不了，包括call，apply，bind。

2 、new 不可用。箭头函数不能使用 new 关键字来实例化对象，不然会报错。

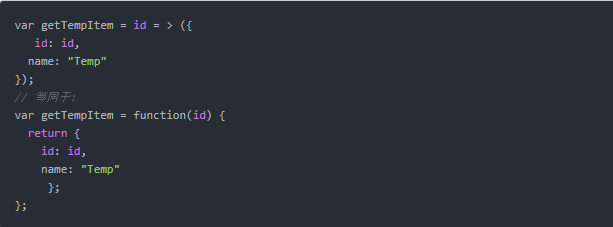
3、this 不可变。函数内置 this 不可变，在函数体内整个执行环境中为常量。

4、没有arguments对象。更不能通过arguments对象访问传入参数。只能使用显式命名或其他ES6新特性来完成。箭头函数没有arguments，caller，callee

5、箭头函数没有原型属性

6、箭头函数不能作为Generator函数，不能使用yield关键字

**7、箭头函数若要返回自定义对象的话，就必须用小括号把该对象括起来先**



用小括号包含大括号则是对象的定义，而非函数主体

8、多重箭头函数就是一个高阶函数，相当于内嵌函数

const add = x => y => y + x;

//相当于

function add(x){

return function(y){

return y + x;

};

}

**注意：箭头函数this的指向 obj::fn3 提案中**

**单点登录(SingleSignOn，SSO)**，就是通过用户的一次性鉴别登录。当用户在身份认证服务器上登录一次以后，即可获得访问单点登录系统中其他联邦系统和应用软件的权限，同时这种实现是不需要管理员对用户的登录状态或其他信息进行修改的，这意味着在多个应用系统中，用户只需一次登录就可以访问所有相互信任的应用系统。这种方式减少了由登录产生的时间消耗，辅助了用户管理，是目前比较流行的。

单点登录(SingleSign-On，SSO)是一种帮助用户快捷访问网络中多个站点的安全通信技术。单点登录系统基于一种安全的通信协议，该协议通过多个系统之间的用户身份信息的交换来实现单点登录。使用单点登录系统时，用户只需要登录一次，就可以访问多个系统，不需要记忆多个口令密码。单点登录使用户可以快速访问网络，从而提高工作效率，同时也能帮助提高系统的安全性。

**SDK** 就是 Software Development Kit 的缩写，中文意思就是“软件开发工具包”。这是一个覆盖面相当广泛的名词，可以这么说：辅助开发某一类软件的相关文档、范例和工具的集合都可以叫做“SDK”。

**“API”**，也就是 Application Programming Interface，其实就是操作系统留给应用程序的一个调用接口，应用程序通过调用操作系统的 API 而使操作系统去执行应用程序的命令（动作）。

**DLL**（又是一个缩写，感觉 IT 这个行业里三字头缩写特别多），即 Dynamic Link Library（动态链接库）。我们经常会看到一些 .dll 格式的文件，这些文件就是动态链接库文件，其实也是一种可执行文件格式。跟 .exe 文件不同的是，.dll 文件不能直接执行，他们通常由 .exe 在执行时装入，内含有一些资源以及可执行代码等。其实 Windows 的三大模块就是以 DLL 的形式提供的（Kernel32.dll，User32.dll，GDI32.dll），里面就含有了 API 函数的执行代码。为了使用 DLL 中的 API 函数，我们必须要有 API 函数的声明（.H）和其导入库（.LIB），函数的原型声明不难理解，那么导入库又是做什么用的呢？我们暂时先这样理解：导入库是为了在 DLL 中找到 API 的入口点而使用的。

所以，为了使用 API 函数，我们就要有跟 API 所对应的 .H 和 .LIB 文件，而 SDK 正是提供了一整套开发 Windows 应用程序所需的相关文件、范例和工具的“工具包”。到此为止，我们才真正的解释清楚了 SDK 的含义。

**js遵循同源策略，即同协议，同域名，同端口号，否则都算跨域。**

**iframe的优缺点及改进方法**

**iframe的缺点**

1、页面样式调试麻烦，出现多个滚动条；

2、浏览器的后退按钮失效；

3、过多会增加服务器的HTTP请求；

4、小型的移动设备无法完全显示框架；

5、产生多个页面，不易管理；

6、不容易打印；

7、代码复杂，无法被一些搜索引擎解读。

**iframe的优点：**

1.iframe能够原封不动的把嵌入的网页展现出来。

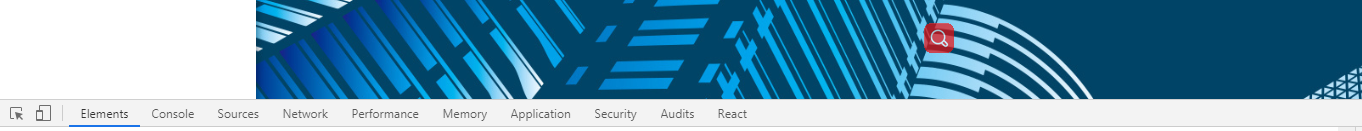
2.如果有多个网页引用iframe，那么你只需要修改iframe的内容，就可以实现调用的每一个页面内容的更改，方便快捷。

3.网页如果为了统一风格，头部和版本都是一样的，就可以写成一个页面，用iframe来嵌套，可以增加代码的可重用。

4.如果遇到加载缓慢的第三方内容如图标和广告，这些问题可以由iframe来解决。

5.重载页面时不需要重载整个页面，只需要重载页面中的一个框架页(减少了数据的传输，增加了网页下载速度)

**SEO（Search Engine Optimization）：**汉译为搜索引擎优化。是一种方式：利用搜索引擎的规则提高网站在有关搜索引擎内的自然排名。目的是：为网站提供生态式的自我营销解决方案，让其在行业内占据领先地位，获得品牌收益；SEO包含站外SEO和站内SEO两方面；为了从搜索引擎中获得更多的免费流量，从网站结构、内容建设方案、用户互动传播、页面等角度进行合理规划，还会使搜索引擎中显示的网站相关信息对用户来说更具有吸引力。



**元素面板：**使用元素面板可以自由的操作 DOM 和 CSS 来迭代布局和设计页面。检查和调整页面、编辑样式、编辑DOM。

**控制台面板：**在开发期间，可以使用控制台面板记录诊断信息，或者使用它作为 shell 在页面上与 JavaScript 交互。使用控制台面板、命令行交互

**源代码面板：**在源代码面板中设置断点来调试 JavaScript ，或者通过 Workspaces（工作区）连接本地文件来使用开发者工具的实时编辑器。断点调试、调试混淆的代码、使用开发者工具的workspaces工作区)进行持久化保存

**网络面板:** 使用网络面板了解请求和下载的资源文件并优化网页加载性能。网络面板基础、资源时间轴、网络带宽限制

**性能面板：**使用时间轴面板可以通过记录和查看网站生命周期内发生的各种事件来提高页面的运行时性能。

**内存面板：**如果需要比时间轴面板提供的更多信息，可以使用“配置”面板，例如跟踪内存泄漏。JavaScript CPU 分析器、存堆区分析器

**应用面板：**使用应用检查加载的所有资源，包括 IndexedDB 与 Web SQL 数据库，本地和会话存储，cookie ，应用程序缓存，图像，字体和样式表

**安全面板：**使用安全面板调试混合内容问题，证书问题等等。

Network : 网络面板隐藏overview Overview: 不需要看时间轴信息时隐藏



1. Overview: 不需要看时间轴信息时隐藏

2. 过滤器: 可以输入字符串或正则表达式，过滤请求，Ctrl + Space显示所有可能的关键字

3. 请求表：

在表头上右键可以添加列（我经常添加Method）

- initiator列: 显示调用堆栈信息，显示哪个脚本的哪一行触发了请求。

- Response Headers: 控制响应头的显示

**你不知道的Chrome调试工具技巧**

1. $0

在 Chrome 的 Elements 面板中，$0 是当前我们选中的 html 节点的引用。

理所当然，$1 就是我们上一次选择的节点的引用，$2 是在那之前选择的节点的引用，等等。一直到 $4

2. $$

返回的是一个节点的 数组

3.$\_ 是对上次执行的结果的引用。

4. $i

在 Chrome插件:Console Importer 的帮助之下，你可以快速的在 console 中引入和把玩一些 npm 库。

直接运行例如 $i('lodash') 或者 $i('moment') 几秒钟之后，就可以获取到 lodash / momentjs 了。

5. copy(...)

6. Store as global

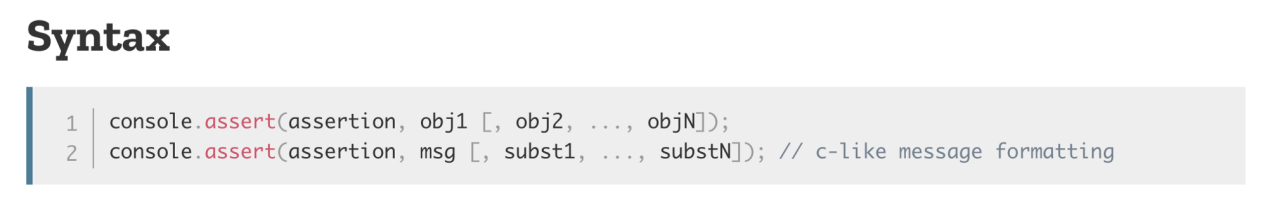
可以轻松将它转换成一个全局变量：只需右击，并且选择 “Store as global variable”(保存为全局变量) 这个选项。

7. 保存(Stack trace)堆栈跟踪

8. Copy HTML (最快的方式)

可能你知道右击或者点击在 html 元素边上的省略号(...)就能将它 copy 到操作系统剪贴板中 但你同样可以用非常古老的 [ctrl] + [c] + [ctrl] + [v] 大法来实现同样的效果！

9. console.assert



当我们传入的第一个参数为假时，console.assert打印跟在这个参数后面的值。当你需要在特殊情况下打印一些信息的时候这很有用 - 现在你可以在不使用if表达式的情况下做到这件事情。另外，你还可以白白得到一个堆栈的跟踪信息 ；-）

10. console.table

令人惊讶的是，这一个小技巧在开发者中并没有多少人知道。如果你有一个数组(或者是类数组的对象，或者就是一个对象)，你可以使用 console.table 方法以一个漂亮的表格的形式将它打印出来。它不仅会根据数组中的对象的所有属性去计算出表中的列名，而且这些列都是可以 缩放 甚至... 排序 ！😳

当列太多的时候，就可以通过第二个参数，只传入你想展示的列名。

11. console.dir

最受欢迎的 console.log 方法会将数据以格式化的形式打印出来，大多数的情况下默认的格式化很有用。但是有时候那可能不是你想看到的 - 最典型的例子就是：你想要打印一个 DOM 节点。

console.log 会将这个交互式的元素渲染成像是从 Elements 中剪切出来的一样。如果说你想要查看这个节点所关联到的真实的 JavaScript 对象呢？并且想要查看他的属性等等？

这样的情况下，如果需要更加直接表现形式来展示数据，就可以使用 console.dir 。

12. 通过 'h' 来隐藏

可以按下 'h' 来隐藏在元素面板中被你选中的元素。再次按下 'h' 使它出现。某些时候这是很有用的，例如你想截图，但是又不想里面包含一些敏感信息。

13. 拖动 & 放置 元素

当你想要看看页面某部分在 DOM 树的不同位置将如何显示的时候，只需拖动并且放置它(到指定的位置)，就像在你机器上其他任何地方一样

14. 或者使用 control !(按钮)

如果只是想在 DOM 结构中往上一点或者往下一点的移动当前被你选中的元素，你可以使用 [ctrl] + [⬆] / [ctrl] + [⬇] ( [⌘] + [⬆] / [⌘] + [⬇] on Mac).

15. 它是一个基础编辑器

此刻我们可以拖动，放置，编辑，复制 (当然，以及使用 [ctrl] + [v] 来粘贴),

所以一旦我们在 Elements 面板中弄乱了我们的 HTML 结构，我们不应该感到惊讶。

在任何文本/图形编辑器中的标准是什么：

使用 [ctrl] + [z] ([⌘] + [z] on Mac) 撤销我们的任何改动。

使用 [ctrl] + [shift] + [z] 重新编辑我们的任何修改。

16. 打印出与预期不符的对象

17. 开始使用 Command (命令) 菜单 (如果你还没有用过的话)

在 Chrome 的调试打开的情况下 按下 [ Ctrl]+[Shift]+[P] (or [⌘]+[Shift]+[P] on Mac)

或者使用DevTools 的 dropdown 按钮的这个选项:

18.截屏，大小通吃

Screen (命令)

19.快速切换面板

Layout(命令)

20.快速切换主题

theme(命令)

21. Console 是异步的

22.用你的异步console 来看一些更酷的东西

26. 给 logs 加上时间戳

如果想要给你的应用中发生的事件加上一个确切的时间记录，开启 timestamps 。你可以在设置(在调试工具中的 ⋮ 下拉中找到它，或者按下 F1 )中来开启或者使用 Commands Menu

27.监测执行时间

相比在所有事件上展示一个时间戳，你或许对脚本中特殊的点之间的时间跨度更加感兴趣

对于这样的情况，我们可以采用一对有效的 console 方法

console.time() — 开启一个计时器

console.timeEnd() — 结束计时并且将结果在 console 中打印出来

28. Custom Formatters 自定义转换器

在大多数的情况下 DevTools 的 console 默认的对象的转化的方式都很有用，但是有时候你期望与众不同。默认的方式总是很 "默认" - 所以我们自定义自己输出对象的方式。做这个事情的函数被称为 Custom Formatter 在我们写下一个之前，需要在 DevTools 的设置 (在 DevTools 的 ⋮ 下拉框找到设置，或者按下 F1 ) 中把对应的设置打开

30. Shadow editor 阴影编辑器

可以通过在 Style 面板中点击靠近 box-shadow 属性或 text-shadow属性的阴影方形符号来打开它

31.Timing function editor 定时函数编辑器

也被称为 Cubic bezier(贝塞尔) 编辑器。贝塞尔曲线是一串用来定义 CSS 的动画速度在整个动画过程中如何变化的魔法数值。我们将其定义为 transition-timing-function 或者 animation-timing-function CSS 属性。

32.插入样式规则的按钮

当你把鼠标放在样式选择器的选择区域的最后时，你会看到几个按钮，它们可以让你快速的使用 Color 和 Shadow 编辑器添加 CSS 属性：

text-shadow

box-shadow

color

background-color

...打开相应的编辑器：

33. Conditional breakpoints 条件断点

有时设置的断点被执行太多次了：想想看，有一个对 200 个元素的循环，但你只对第 110 次循环的结果感兴趣，或者只对一些满足其他的特殊条件的结果感兴趣。

这样的情况下，就可以设置一个条件断点。实现：

右击行号并且选择 Add conditional breakpoint...(添加条件断点) 的选项

或者右击一个已经设置的断点并且选择 Edit breakpoint(编辑断点)

然后输入一个执行结果为 true 或者 false 的表达式（它的值并不需要明确的为 true 或者 false 尽管那个弹出框的描述是这样说的）。

在这个表达式中，你可以获取到任何这段代码可以获取到的东西，即这一行的作用域。

现在条件满足的话，断点就会暂停代码的执行

34.The ninja（忍者） console.log

得益于条件断点，现在可以开始灵活使用这个技术。

因为：

每一个条件都必须经过判断 - 也就是 - 运行 - 每当应用执行到这一行。

并且如果条件返回的是falsy的值(例如. undefined )，它并不会暂停..

所以，与其在源码的不同地方去添加 console.log / console.table / console.time 等等，不如直接使用条件判断来将它们 “连接” 到 Source 面板中。它们不会停止，会一直执行，并且当你不再需要它们的时候，有一个地方( Breakpoints section )会列出它们。点两下鼠标就可以把所有的都移除，就像一堆忍者一样消失！

35. queryObjects function 对象查询方法

36. monitor functions 镜像函数

monitor 是 DevTools 的一个方法, 它能够让你 spy(潜入) 到任何 "function calls(方法的调用)" 中：每当一个 spied 被潜入 的方法运行的时候，console 控制台会把它的实例打印出来，包含函数名以及调用它的参数。

37. 在两个最近使用的 DevTools 的展示位置之间切换

ctrl + shift + D底部的 dock或者右部的dock(开发工具的显示位置是下边还是右边)

39. Increment/decrement 递增/递减

这个对 Style 部份最有用：通过使用带有或者不带有修饰键的上/下 箭头按键 你可以很轻松的递增和递减 0.1 , 1 或者 10 这样数值类型的值。

40. monitorEvents函数

41. console.log 基于调用堆栈自动缩进

42. 给你的 console.log 加上 CSS 样式

给你的打印文本加上 %c 然后 console.log 的第二个参数变成了...CSS 规则！你可以利用这一点让你的日志脱颖而出(例如 Facebook 在你打开 console 的时候所做的一样)



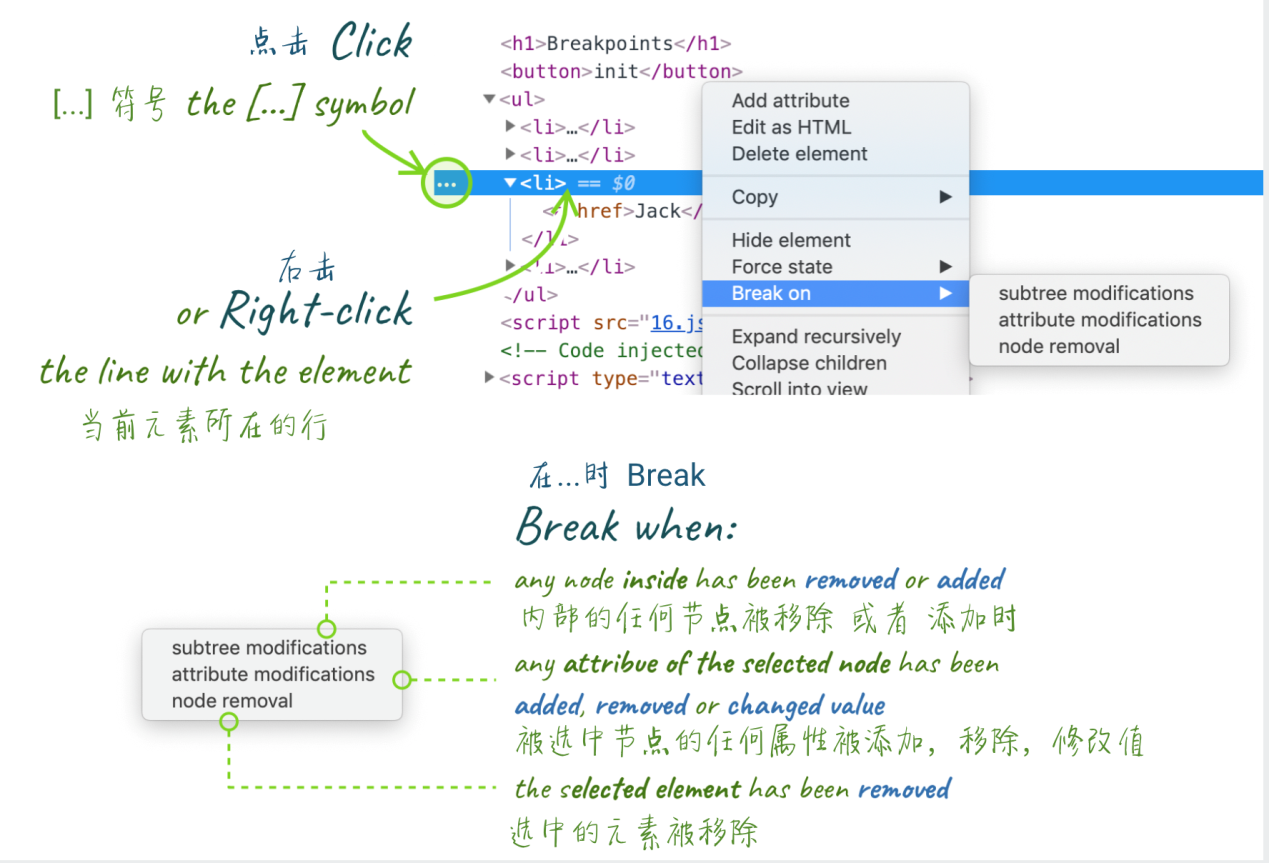
43. replay XHR requests (重新发送 XHR 的请求)

44.在元素面板中展开所有的子节点

相比于点击级联的 ▶ 按钮，不如使用右击节点后的 expand recursively 命令

45. DOM breakpoints (DOM 断点)

某些脚本会修改 DOM 的一部分，不确定修改的是哪部分？甚至不确定是什么时候修改的？添加一个 DOM 断点。你就可以监听到节点被添加或者移除，或属性被改变。



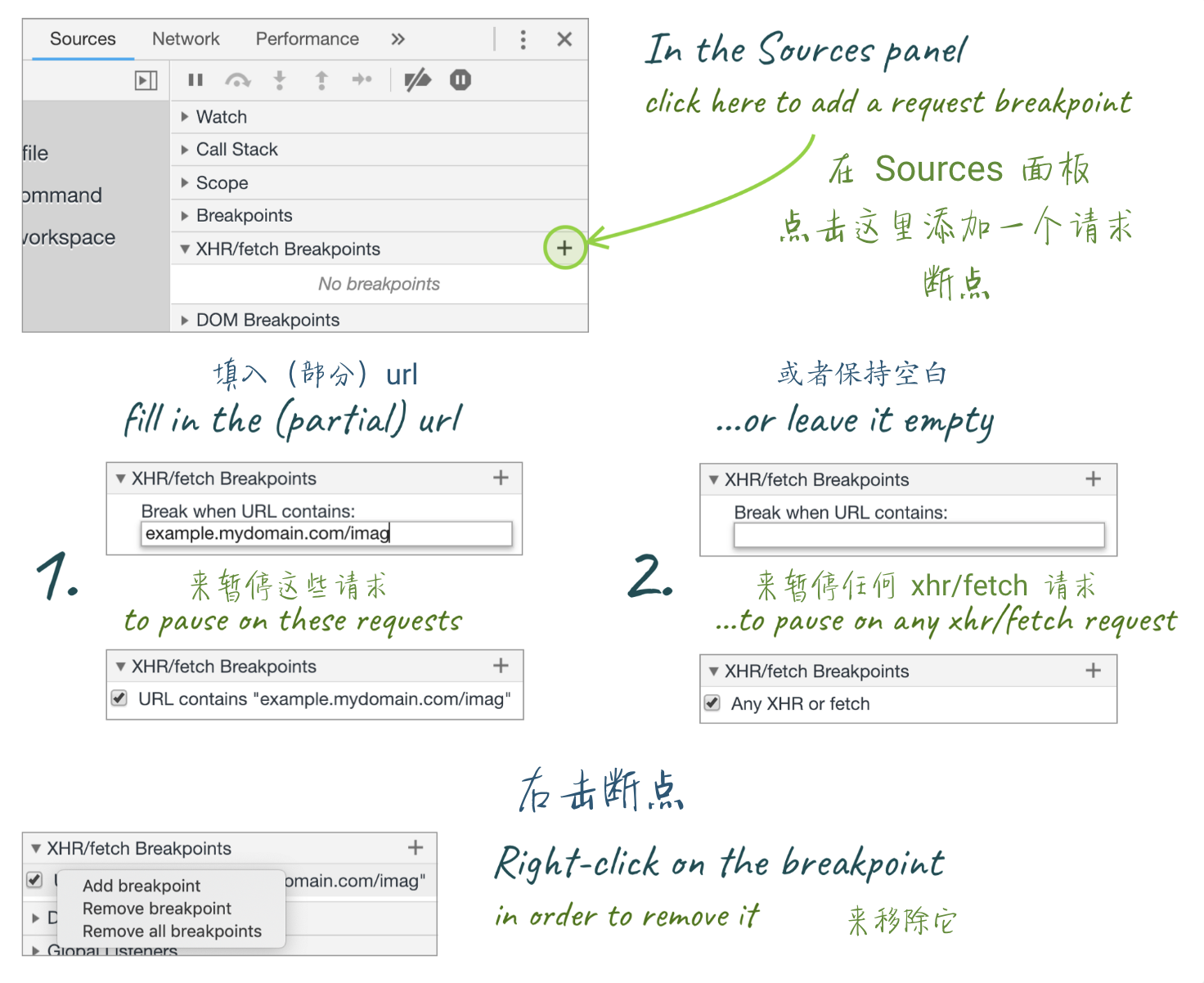
46. MutationObserver （变化观察者）

XHR/fetch 断点

当你想捕获已发送的 “ajax” 请求中的特定时刻，

可以使用 XHR/fetch breakpoint。这些只能在 Source 面板中设置。（我也同样希望在 Network 面板中可以设置，但事实并非如此）

您可以添加部分 URL 作为触发器或监听任何请求：

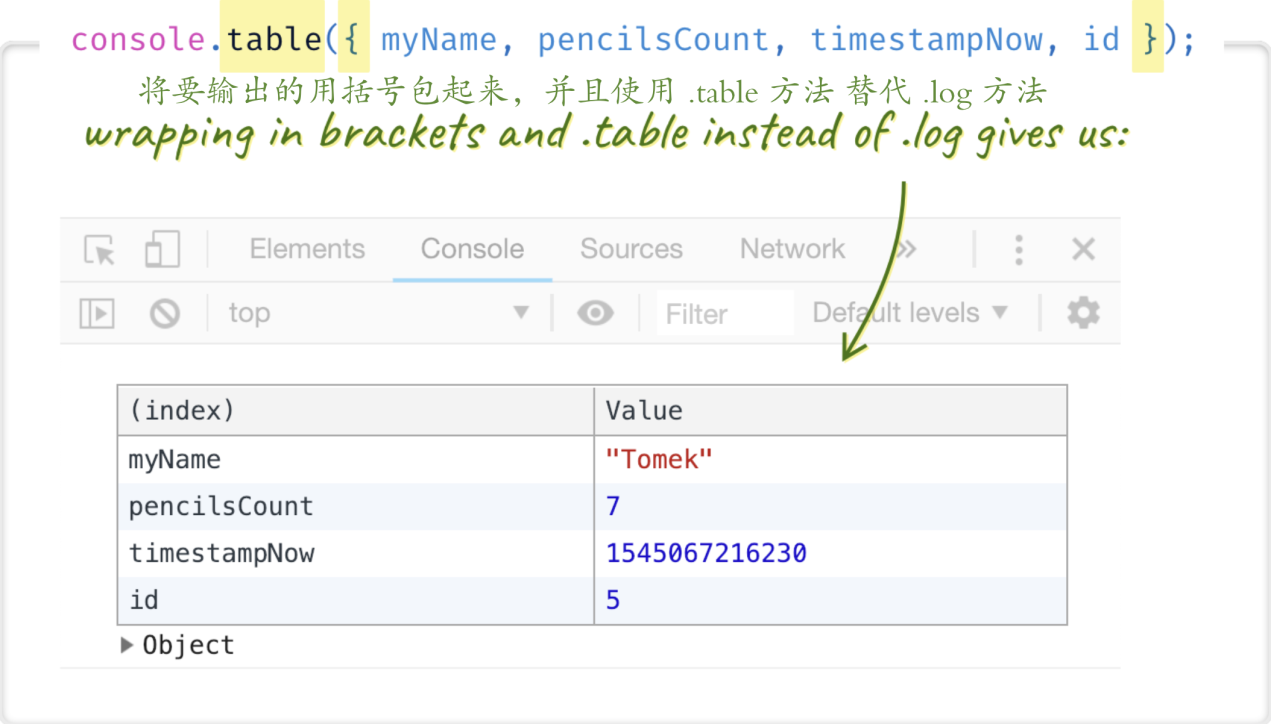


48. For more readable logs (更易读的日志)

为了让它变得更加易读，你可以打印一个对象 - 只需将所有console.log 的参数包装在大括号中。感谢 ECMAScript 2015 中引入了 enhanced object literal(增强对象文字面量) ，所以这（加了个括号）就是你需要做的全部了：



49. Brackets + table = 🤯 （大括号 + 表格 = 🤯 ）



50. The DevTools drawer

Chrome DevTools 有很多部分，被分为 9 个 tab(标签页/选项卡) ( Elements , Console , Sources , Network , 等等...)- 但那仅仅是它的一部分而已！有一组平行的选项卡，被隐藏在主窗口之下。这个组合被称为 Drawer

要访问它，当你在 DevTools（任何选项卡） 中时，按 [esc] 。

再次按 [esc] 隐藏它

61. Hide network overview（ 隐藏 network overview ）

62. Request initiator 显示了调用堆栈信息

63.按照方法过滤请求

65.检查代码 coverage

使用 coverage 面板来获得关于冗余代码的摘要-细节信息。使用 Drawer 菜单或者 Command 菜单来打开它。

66. Check what you’ve changed (检查你修改的内容)

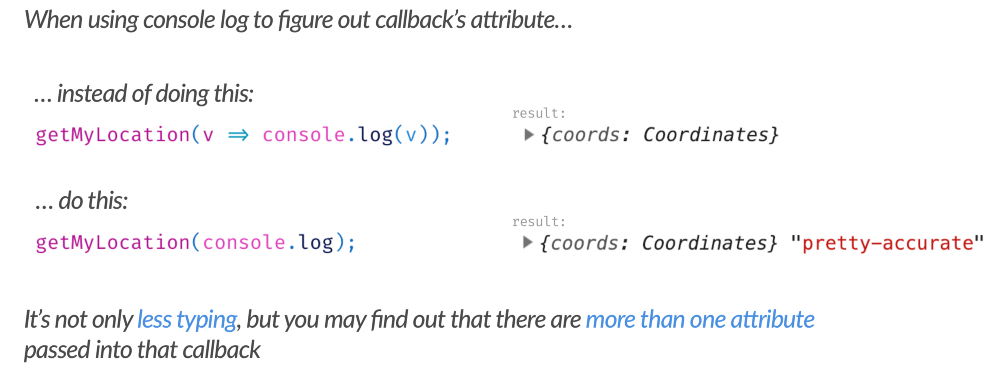
在浏览器中进行设计，调整这些css值，在代码执行的地方进行调试 - 这既方便又有趣。 但在某些时候，你可能希望将已更改的内容与最初加载的样式表进行比较。

为此，可以使用位于 DevTools' Drawer 中的更改工具。

67. Use console.log in callback directly（直接在回调中使用 console.log ）

我经常发现自己处于这样一种情况，就是我不完全确定要将什么传递给回调函数。在这种情况下，我会在里面添加一个 console.log 来检查。

有两种方式来实现：在回调方法的内部使用 console.log 或者 直接使用 consolelog 来作为回调方法。这不仅可以减少输入，而且可能回调接收多于1个参数



69.Use live expressions 使用实时表达式

2个月前，DevTools 在 Console 面板中引入了一个非常漂亮的附加功能，这是一个名为 Live expression 的工具

只需按下 "眼睛" 符号，你就可以在那里定义任何 JavaScript 表达式。 它会不断更新，所以表达的结果将永远，存在 :-)

**#JavaScript# 插件、 组件、类库、框架的区别**

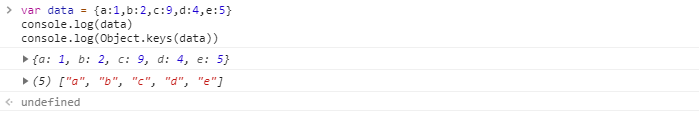
**类库：**提供了一些真实项目开发中常用的方法，这些方法做了一些完善处理，比如兼容处理、细节优化等，方便我们开发和维护。常用的类库有：JQuery、Zepto

**插件：**把项目中某一部分进行插件分装，具备具体的业务逻辑，有针对性。如果项目中有类似需求，直接导入插件代码即可，相关逻辑代码不需要自己在写一遍。常用插件：jquery.drag.js、jquery.validate.min.js、jquery.dialog.js、datepicker日历插件、echarts统计图插件、iscroll、swiper插件

**组件：**类似于插件，但是插件一般只是把JS部分封装，组件不仅分装了JS部分，还有CSS部分，以后再使用的时候，我们直接按照文档使用说明引入CSS/JS文件，搭建对应的结构即可。常用的组件有：Bootstrap、swiper组件

**框架：**比上面的三个都要庞大。它不仅提供了很多常用的方法，而且也可以支持一些插件的扩展（可以把一些插件集成到框架中运行），提供了非常优秀的代码管理设计思想。框架有：react、vue、react-native

Object.keys()方法会返回一个由一个给定对象的自身可枚举属性组成的数组。



**Map和Set**

JavaScript的默认对象表示方式{}可以视为其他语言中的Map或Dictionary的数据结构，即一组键值对。

但是JavaScript的对象有个小问题，就是键必须是字符串。但实际上Number或者其他数据类型作为键也是非常合理的。

为了解决这个问题，最新的ES6规范引入了新的数据类型Map。

**Map**

Map是一组键值对的结构，具有极快的查找速度。

**var** m = **new** Map([['Michael', 95], ['Bob', 75], ['Tracy', 85]]);

m.get('Michael'); *// 95*

初始化Map需要一个二维数组，或者直接初始化一个空Map。Map具有以下方法：

**var** m = **new** Map(); *// 空Map*

m.set('Adam', 67); *// 添加新的key-value*

m.set('Bob', 59);

m.has('Adam'); *// 是否存在key 'Adam': true*

m.get('Adam'); *// 67*

m.**delete**('Adam'); *// 删除key 'Adam'*

m.get('Adam'); *// undefined*

由于一个key只能对应一个value，所以，多次对一个key放入value，后面的值会把前面的值冲掉：

var m = new Map();

m.**set**('Adam', 67);

m.**set**('Adam', 88);

m.get('Adam'); // 88

### Set

Set和Map类似，也是一组key的集合，但不存储value。由于key不能重复，所以，在Set中，没有重复的key。

要创建一个Set，需要提供一个Array作为输入，或者直接创建一个空Set：

**var** s1 = **new** Set(); *// 空Set*

**var** s2 = **new** Set([1, 2, 3]); *// 含1, 2, 3*

重复元素在Set中自动被过滤：

var s = new **Set**([1, 2, 3, 3, '3']);

s; // **Set** {1, 2, 3, "3"}

注意数字3和字符串'3'是不同的元素。

通过add(key)方法可以添加元素到Set中，可以重复添加，但不会有效果：

s.add(4);

s; // **Set** {1, 2, 3, 4}

s.**add**(4);

s; // 仍然是 **Set** {1, 2, 3, 4}

通过delete(key)方法可以删除元素：

var s = new **Set**([1, 2, 3]);

s; // **Set** {1, 2, 3}

s.**delete**(3);

s; // **Set** {1, 2}

### 小结

Map和Set是ES6标准新增的数据类型，请根据浏览器的支持情况决定是否要使用。

**谈谈iframe的优缺点**

iframe是一种框架，也是一种很常见的网页嵌入方式iframe的优点：

1.iframe能够原封不动的把嵌入的网页展现出来。

2.如果有多个网页引用iframe，那么你只需要修改iframe的内容，就可以实现调用的每一个页面内容的更改，方便快捷。

3.网页如果为了统一风格，头部和版本都是一样的，就可以写成一个页面，用iframe来嵌套，可以增加代码的可重用。

4.如果遇到加载缓慢的第三方内容如图标和广告，这些问题可以由iframe来解决。

iframe的缺点：

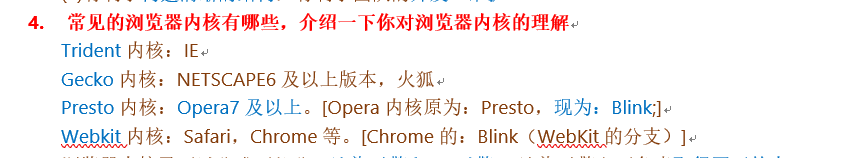
1.会产生很多页面，不容易管理。

2.iframe框架结构有时会让人感到迷惑，如果框架个数多的话，可能会出现上下、左右滚动条，会分散访问者的注意力，用户体验度差。

3.代码复杂，无法被一些搜索引擎索引到，这一点很关键，现在的搜索引擎爬虫还不能很好的处理iframe中的内容，所以使用iframe会不利于搜索引擎优化。

4.很多的移动设备（PDA 手机）无法完全显示框架，设备兼容性差。

5.iframe框架页面会增加服务器的http请求，对于大型网站是不可取的。分析了这么多，现在基本上都是用Ajax来代替iframe，所以iframe已经渐渐的退出了前端开发

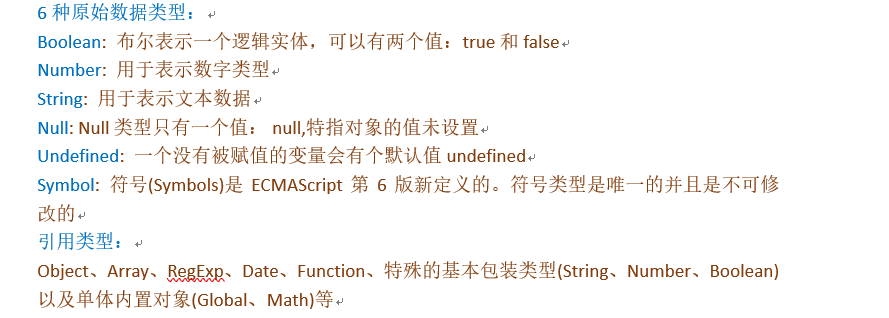
****

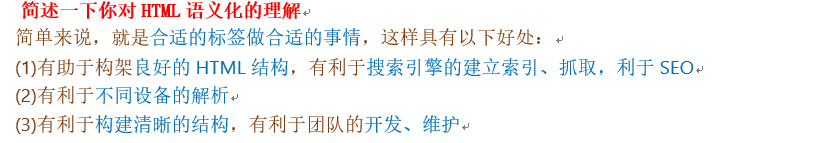
**IE : Trident**

**Firefox: Gecko**

Safari:webkit内核

Blink:Chrome（28及往后版本）、Opera（15及往后版本）和Yandex浏览器中使用

****

****

**http 常见状态码：**

200("OK")

500 ("Internal Server Error")

404 ("Not Found")

1XX：通知

2XX: 成功

3XX 重定向

4XX：客户端错误

5XX 服务端错误

**1. 什么是BFC？**

BFC全称为Block Formatting Context，即“块级格式化上下文”，它是页面中相对独立的一块渲染区域，它决定了内部的子元素如何进行摆放和定位，以及区域内部元素和区域外部元素之间的相互作用关系。

**2. BFC有什么特点？**

当一个元素容器创建BFC后，主要有以下表现特点：

BFC可以包含浮动元素（闭合浮动）

BFC所确定的区域不会与外部浮动元素发生重叠

位于同一BFC下的相邻块级子元素在垂直方向上会发生margin重叠

位于不同BFC下的相邻元素之间不会发生margin重叠

将以上特点一言以蔽之，即BFC在页面上是一个封闭的区域，如同“结界”一般。即便是内部的浮动元素也无法脱离该区域。该区域内部的子元素无法影响区域外部，同时也不受外部影响。

**3. 如何触发/创建BFC？**

若某个元素满足以下任一条件，则会对其创建BFC：

<html>根元素

float的值不为none

overflow的值为auto、scroll或hidden

display的值为table-cell、table-caption或inline-block

position的值为fixed或absolute

**4. BFC的常见用途**

闭合浮动

阻止margin重叠

自适应流体布局:BFC最强大的用途其实是用于自适应流体布局，这是基于BFC所确定的区域不会与外部浮动元素发生重叠的特性实现的

**HTML DOM addEventListener() 方法**

addEventListener() 方法用于向指定元素添加事件句柄。

提示： 使用 removeEventListener() 方法来移除 addEventListener() 方法添加的事件句柄。

**浏览器支持情况：**

注意： Internet Explorer 8 及更早IE版本不支持 addEventListener() 方法，，Opera 7.0 及 Opera 更早版本也不支持。 但是，对于这些不支持该函数的浏览器，你可以使用 attachEvent() 方法来添加事件句柄

## 语法

element.addEventListener(event, function, useCapture)

## 参数值

|  |  |
| --- | --- |
| **参数** | **描述** |
| event | 必须。字符串，指定事件名。  **注意:** 不要使用 "on" 前缀。 例如，使用 "click" ,而不是使用 "onclick"。   **提示：** 所有 HTML DOM 事件，可以查看我们完整的 [HTML DOM Event 对象参考手册](https://www.runoob.com/jsref/dom-obj-event.html)。 |
| function | 必须。指定要事件触发时执行的函数。   当事件对象会作为第一个参数传入函数。 事件对象的类型取决于特定的事件。例如， "click" 事件属于 MouseEvent(鼠标事件) 对象。 |
| useCapture | 可选。布尔值，指定事件是否在捕获或冒泡阶段执行。  可能值:   * true - 事件句柄在捕获阶段执行 * false- false- 默认。事件句柄在冒泡阶段执行 |

IE中：

target.attachEvent(type, listener);

target： 文档节点、document、window 或 XMLHttpRequest。

type： 字符串，事件名称，含“on”，比如“onclick”、“onmouseover”、“onkeydown”等。

listener ：实现了 EventListener 接口或者是 JavaScript 中的函数。 例如：document.getElementById("txt").attachEvent("onclick",function(event){alert(event.keyCode);});

移除：

Windows IE的格式如下:

detachEvent(event,function);

**HTML DOM querySelector() 方法**

**定义与用法**

querySelector() 方法返回匹配指定 CSS 选择器元素的第一个子元素 。

注意： querySelector() 方法只返回匹配指定选择器的第一个元素。如果你要返回所有匹配元素，请使用 querySelectorAll() 方法替代。

循环（loop），指的是在满足条件的情况下，重复执行同一段代码。比如，while语句。

迭代（iterate），指的是按照某种顺序逐个访问列表中的每一项。比如，for语句。

遍历（traversal），指的是按照一定的规则访问树形结构中的每个节点，而且每个节点都只访问一次。

递归（recursion），指的是一个函数不断调用自身的行为。比如，以编程方式输出著名的斐波纳契数列。