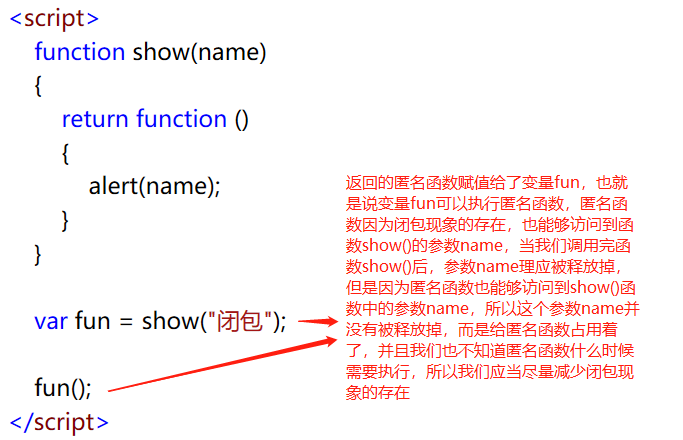
**JavaScript**

**使用JavaScript的注意问题**

1. JavaScript需写在HTML文档当中的<script></script>标签当中，目的是为了区分在HTML页面下的JavaScript脚本
2. JavaScript可以写在三处地方
3. 写在HTML页面的<head></head>标签当中：用于声明变量、函数、类型和为标签事件绑定处理的函数
4. 写在HTML页面的<body></body>标签当中：执行JS脚本
5. 外部自定义一个JavaScript文件，然后通过<script></script>标签的src属性进行文件路径的赋值，即可导入我们所写的JavaScript脚本（外部JS文件的里面不需要写<script></script>标签，因为<script>标签的目的只是在于区分HTML页面下的JavaScript脚本）
6. JavaScript是一种弱类型语言
7. 弱类型语言：在声明一个变量的时候，不需要指定这个变量的类型，也就是不需要明确声明这个变量的内存地址和所占内存的大小，只有当代码运行的时候，才会知道这个变量是占有多大的内存空间，也就是说，如果我们在弱类型语言中定义了一个变量，并在代码未运行前，我们可以随意更改这个变量的类型，因为弱类型语言是只有当程序运行的时候才会编译出变量的内存地址所占内存空间的大小（ JavaScript声明变量用 var ）
8. 强类型语言：在声明一个变量时，必须指定这个变量的类型，也就是指定这个变量的内存地址，并指定了该变量所占内存的大小，如：c#，举个例子：int类型和double类型所占内存是不一样的字节，如果我们单纯的通过隐式转换的话那么编译器就会报错，如果进行强转的话，因为两个类型所占字节的不同，所以可能会造成数据的缺失

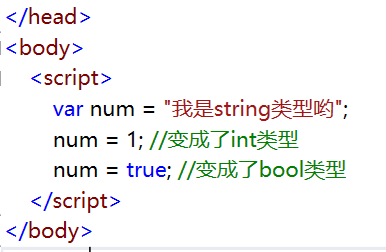
**JavaScript中的闭包和作用域链的概念**

1. 作用域链：当我们声明了一个函数A并声明了一个函数A的参数变量variable，我们又在函数A的内部声明了一个函数B，而这时候函数B却能够访问到未声明在自己方法体中的参数变量variable，这一现象就是作用域链。简而言之，作用域链的作用在于，使得变量的作用域在，当前函数中及当前函数内部所定义的函数中，形成了一个链条
2. 闭包：
3. 因为作用域链的存在，才会产生闭包这种现象
4. 闭包就是一个函数把外部的那些不属于自己的对象也包含（闭合）进来了
5. 举个例子：我们声明了一个函数A并在函数A中有一个参数变量variable，这之后我们又在函数A中定义了一个函数B，但是在函数B的内部也能够访问到并未声明在自己内部的函数A的参数变量variable，这就是闭包的现象
6. 我们应当尽量的少避免闭包的情况的发生（除了声明类的情况），如下图的闭包现象：在函数show()中有一个参数name，并在函数show()的内部又返回了一个匿名函数，而在匿名函数的内部也能够访问到函数show()的参数name，这时候就会产生一个问题，闭包的现象虽然能够让我们少写 些赋值的代码，但是当我们调用了函数show()并执行完了，参数name应该要释放掉内存的占用，但是函数show()却返回了另一个匿名函数，并且该匿名函数却访问到了函数show()所声明的参数name，并且我们也不知道返回的匿名函数需要啥时候需要被执行，所以这个参数变量name的内存其实就是一直在被占用着

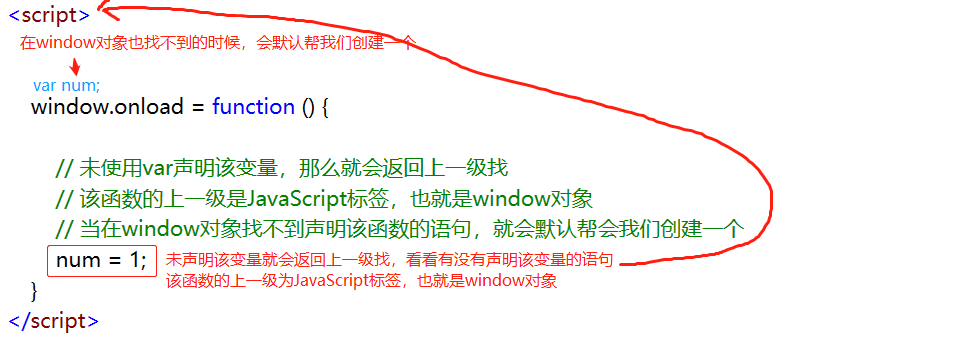


**JavaScript的具体使用**

1. **JavaScript变量的声明**
2. 使用var：因为是弱类型语言，我们在程序运行前可以随意对该变量的类型进行改变

****

1. 不使用var，直接使用变量：当我们直接使用变量的时候，由于闭包的原因，会返回上一级寻找，而我们在JavaScript标签中写的代码（不包含函数内部声明的函数）的上一级就是JavaScript标签（window对象），如果在window对象下也找不到这个变量，那么会默认帮我们创建一个。**但是我们不推荐这种方式**



1. **JavaScript虽然是弱类型语言，并且变量的声明用var，但是，不同的变量的值也对应着不同的类型**

**boolean布尔类型**

**微信截图_20180412214229**

**number数字类型**

**微信截图_20180412214237**

**string字符串类型**

**微信截图_20180412214233**

**object对象类型**

**微信截图_20180412214314**

**Undefined未定义值的类型：**

1. **当我们声明了一个变量，但却没有指定该变量的值，那么这个变量的值就为undefined**
2. **Undefined类型的变量的值就是undefined，不需要我们指定**
3. **Undefined类型的值与Null空对象类型的值null是相等的，都是空**

**微信截图_20180412214241**

**Null空对象类型：**

1. **Null类型需要我们指定变量的值为null才能定义变量类型为Null类型，简而言之，null也是个值，但它是个空的值**
2. **Null空对象类型其实是一个object类型，但是却是一个空的object类型**
3. **Null类型的值就是null，需要我们指定一个null值才行**
4. **Undefined类型的值undefined与Null空对象类型的值null是相等的，都是空**

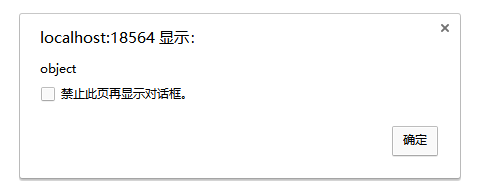
**微信截图_20180412214309**

1. **typeof(变量名)/typeof 变量名：能够获取变量的类型名字，并返回该类型名字的字符串**

**代码规范**

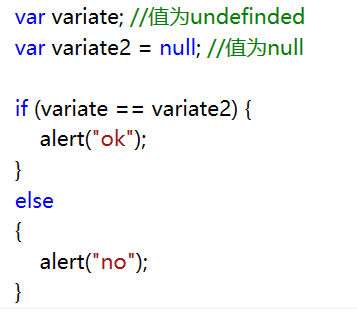
微信截图_20180412214620

**实现效果**

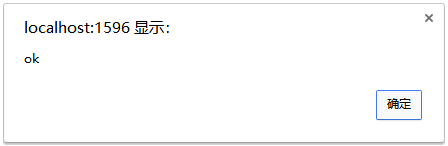


1. **JavaScript中的判等**
2. ==：单纯的判断两个变量的值是否相等，**类型不同也可用于判断**

**代码规范**

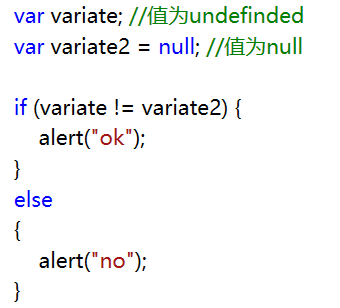


**实现效果**

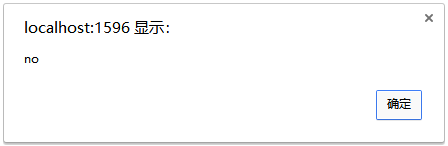


1. !=：单纯的判断两个变量的值是否不想等，**类型不同也可用于判断**

**代码规范**

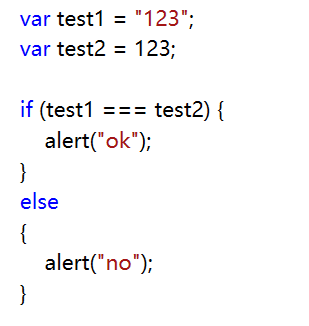


**实现效果**

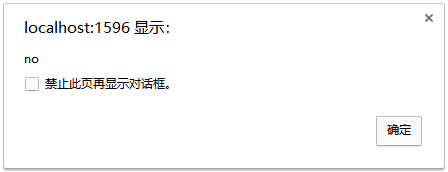


1. ===：先判断两个变量的类型是否相等，然后再判断两个变量的值是否不想等，**类型不同也可用于判断**

**代码规范**

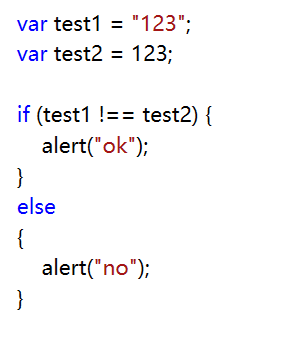
****

**实现效果**

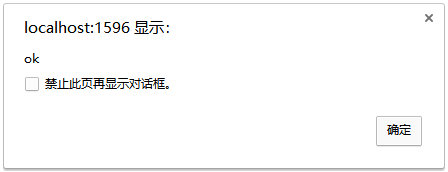
****

1. !==：先判断两个变量的类型是否不想等，然后再判断两个变量的值是是否不想等，**类型不同也可用于判断**

**代码规范**

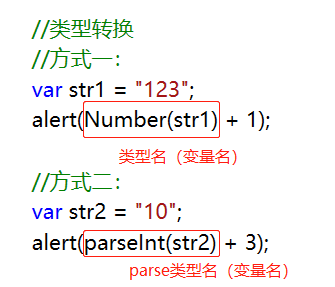
****

**实现效果**

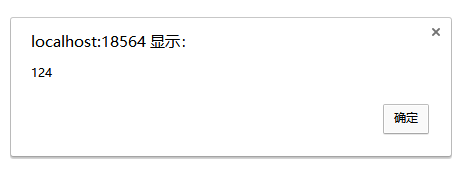
****

1. **类型名（变量名）/parse类型名（变量名）：类型转换**

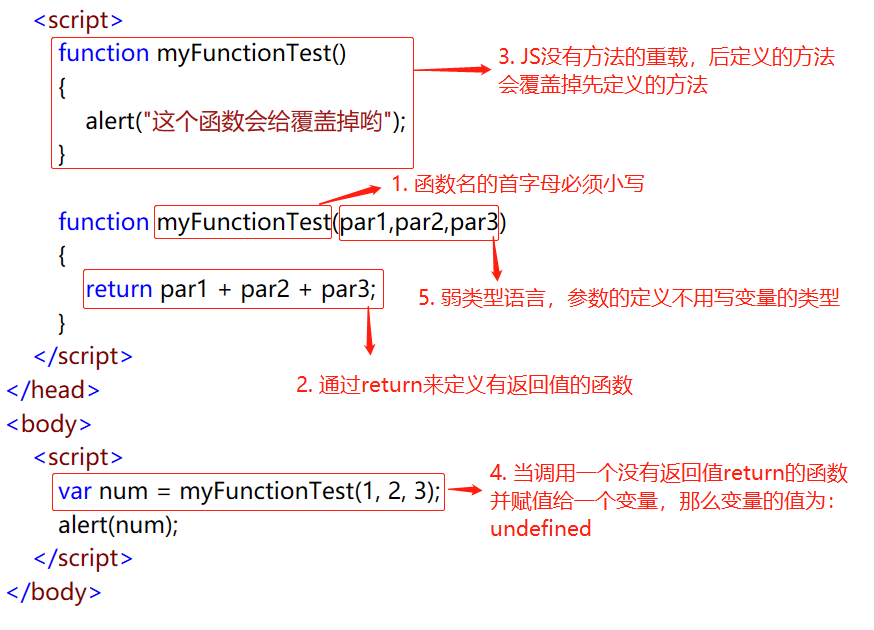
**代码规范**

****

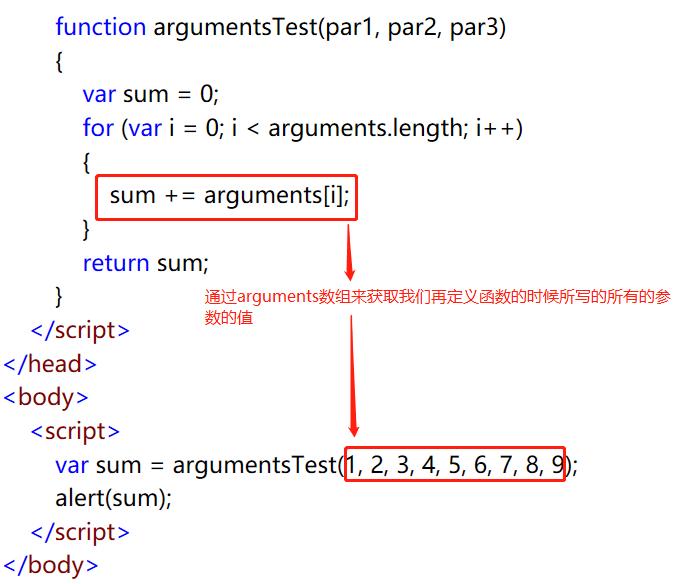
**实现效果**

****

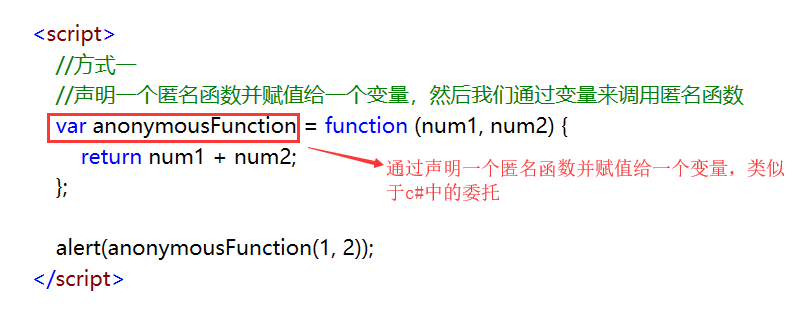
1. **函数的定义：function()**
2. 函数的注意事项：
3. 函数名的首字母必须小写
4. 通过return来定义有返回值的函数
5. JavaScript中的函数没有方法的重载，由于是由上至下的执行顺序，后定义的函数会覆盖掉先定义的函数
6. 当调用没有返回值的函数并赋值给一个变量时，变量的值为：undefined
7. 由于JavaScript是弱类型语言，所以我们在函数的参数中不用写参数的类型，只用写参数的名
8. **当我们不写参数的时候，在调用函数我们可以随意赋值参数的个数，并且在函数中可以通过arguments数组来获取我们在调用函数的时候所写的所有的参数，类似于c#的 params 可变参数**



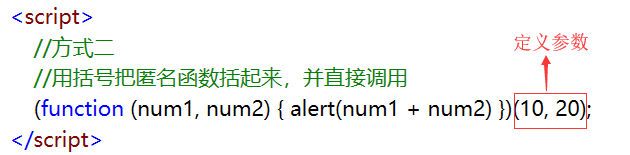
1. 在函数中，通过arguments数组可以获取在调用函数的时候我们所写的所有的参数：
2. 函数中，arguments数组的使用



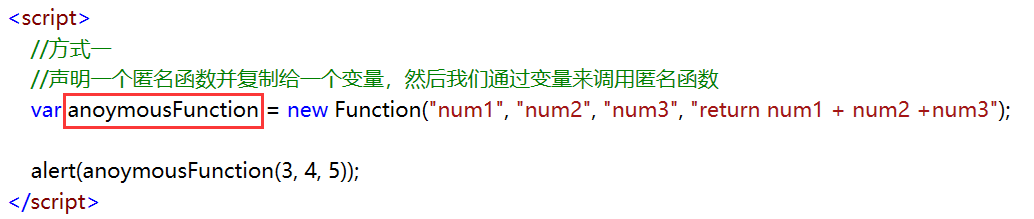
1. **匿名函数的定义**
2. fucntion (参数1 ，参数2){ 方法体 } 来定义一个匿名函数：
3. 方式一：把匿名函数赋值给一个变量，然后通过变量来调用



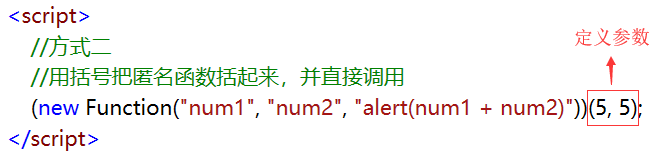
1. 方式二：直接调用自己定义的匿名函数



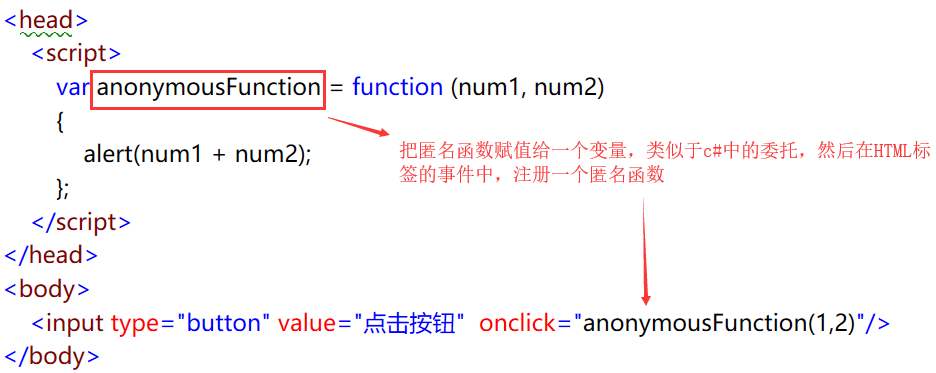
1. new Function ("参数1" ，"参数2" ，"参数…" ，"方法体") 来定义一个匿名函数：
2. 方式一：把匿名函数赋值给一个变量，然后通过变量来调用



1. 方式二：直接调用自己定义的匿名函数



1. 在HTML一个标签的事件中，注册一个匿名函数：



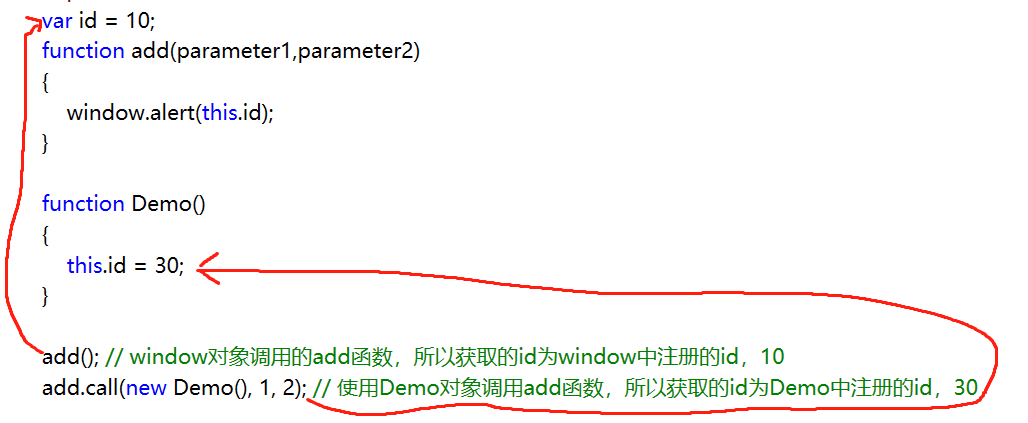
1. **JS中call()和apply()的使用**

-> call和apply所实现的效果是相同的，只是参数的写法不同

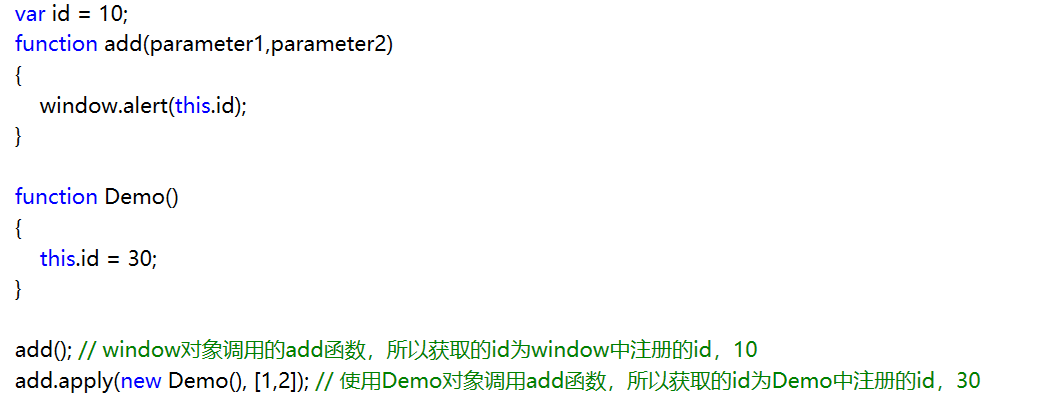
-> call/appy能够在不改变 函数/类 的源代码的前提下，把调用该函数的 类/函数 内部的this指向为我们所定义的对象

-> 当 类A 中有一段代码：使用 类B 调用了call/apply函数，并且指定了object为 类A 中的this，那么每当我们创建一次 类A 的实例，都会执行上述所说的代码，而执行了那一段代码后，我们指定 类B 的this转换为 类A 中的this，所以 类B 中的成员也会动态注册至 类A 之中（因为 类B 中的this指向为 类A ），所以只要这段代码存在于 类A 的内部中，这种改变针对于 类A 的实例是**永久性**的，这样也就**实现了继承**

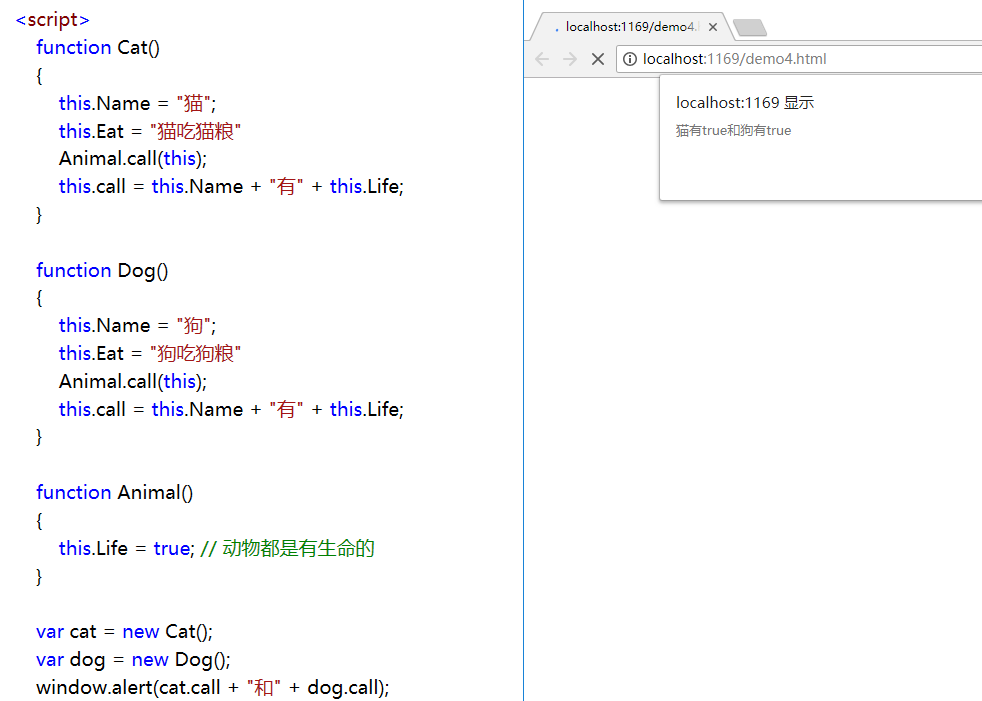
1. 类/函数.call( object，parameter1，……，parameterN )：将当前 函数/类 内部的this所代表的对象设置为我们所指定的object后再调用当前 函数/类



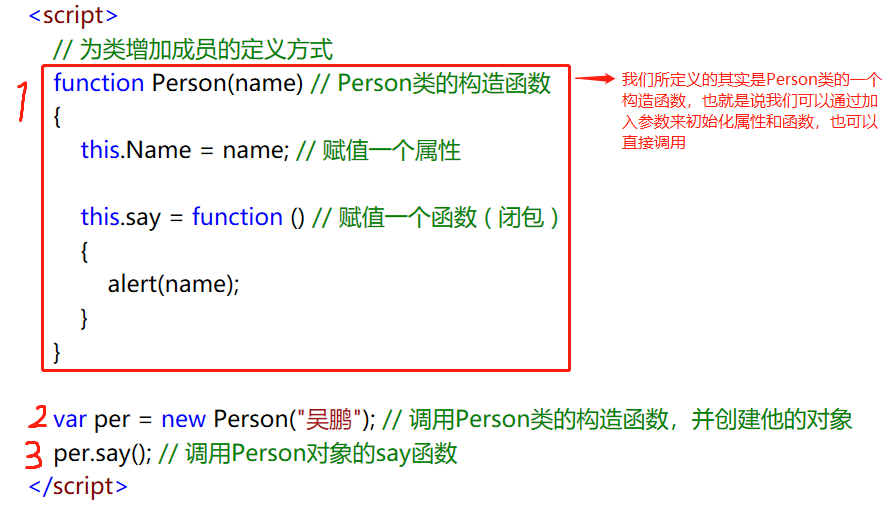
1. 类/函数.apply( object，parameters[] )：将当前 函数/类 内部的this所代表的对象设置为我们所指定的object后再调用当前 函数/类



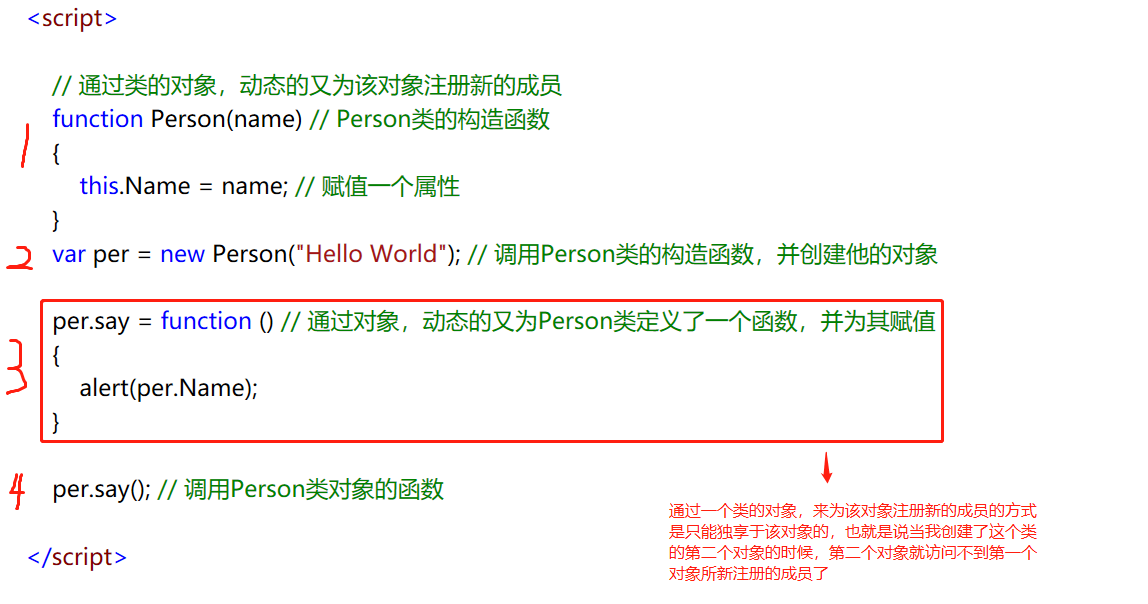
1. 使用call/appy实现继承



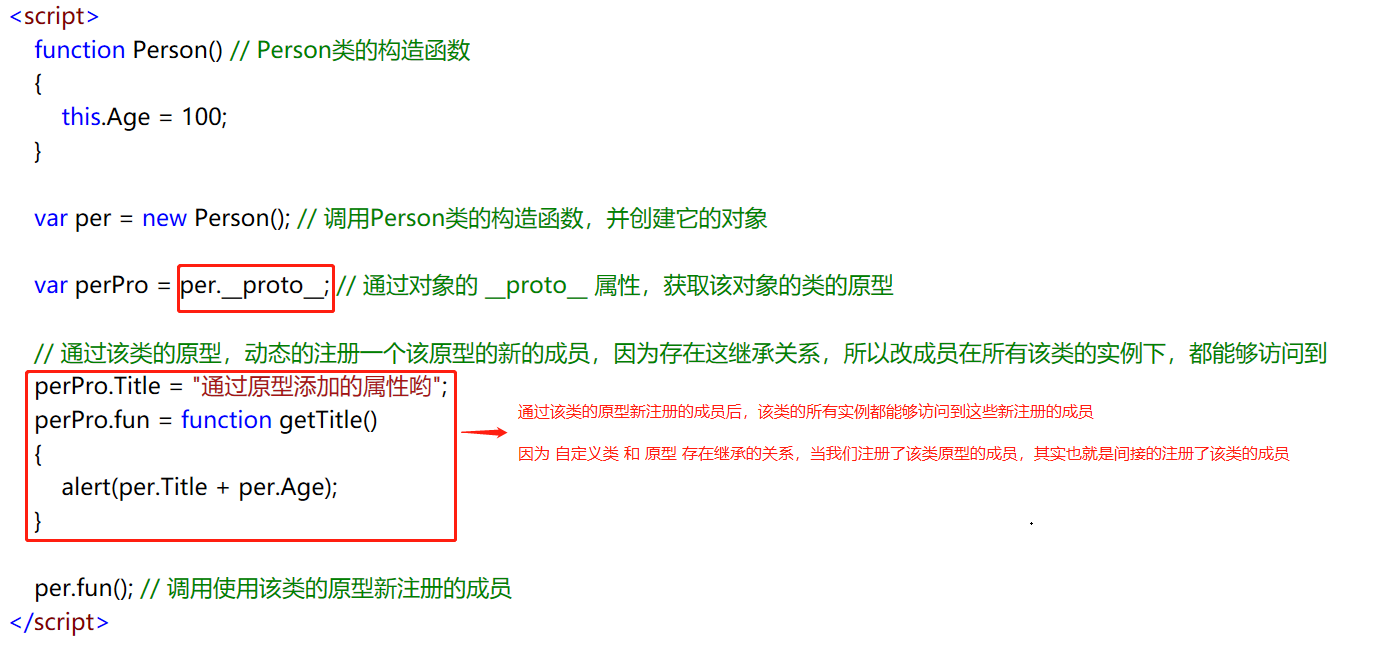
1. **JavaScript模拟面向对象**
2. 定义类的注意事项：
3. JavaScript中定义类的方式和定义函数一样，所以注意定义类的时候**首字母必须大写**
4. 定义一个类的基本方式：
5. 其实我们所定义的，是这个类的构造函数，我们可以通过参数来为类中的 属性、函数 进行初始化
6. 如果该类的构造函数有一些具体的实现，如：return返回值，那么我们可以直接把该类当作一个函数来调用，也可以创建该类的对象然后通过对象访问该类的一些属性和函数



1. 通过类的对象，为该对象动态的注册新的成员，并且该对象独享于这些成员：
2. 我们需要先创建对象，才能通过对象去动态的注册新增的成员
3. 当我们创建了该类的对象，并通过对象去动态的注册新增的成员，但是这些成员是只针对该对象所享有的，也就是说，当我们创建第二个该对象，就不能用第一个对象所初始化的属性、函数的值，需另外再初始化（**除非是用原型**）



1. **JS中原型的应用**
2. 原型的详解：
3. 原型在JS中用于描述一个类或对象的原来的类型，可以理解为用于描述一个类或对象的父类
4. 一个类与该类的原型，存在着继承的关系，也就是说当我们获取到了该类的原型并为该类的原型动态的注册新的成员，那么该类的所有实例也能够访问到原型所新注册的成员
5. 通过原型，我们就能实现JS的继承，也可以让我们随便的为类去扩展成员
6. 通过对象的：\_\_proto\_\_ 属性来获取该类的原型



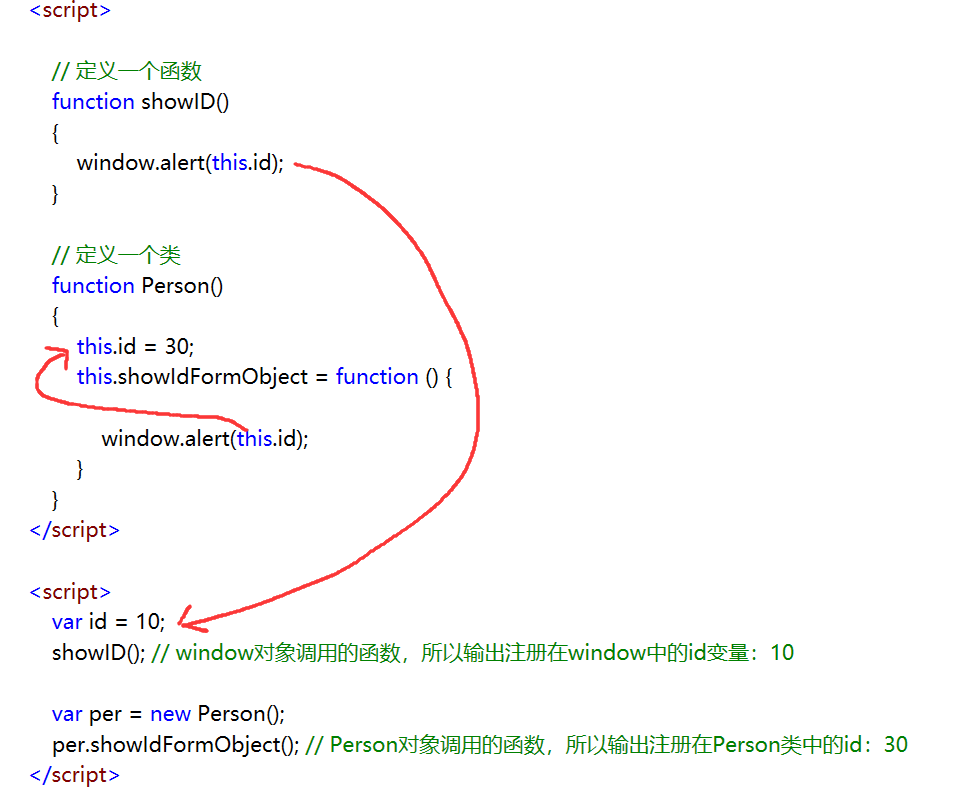
1. 通过类的：prototype 属性来获取该类的原型



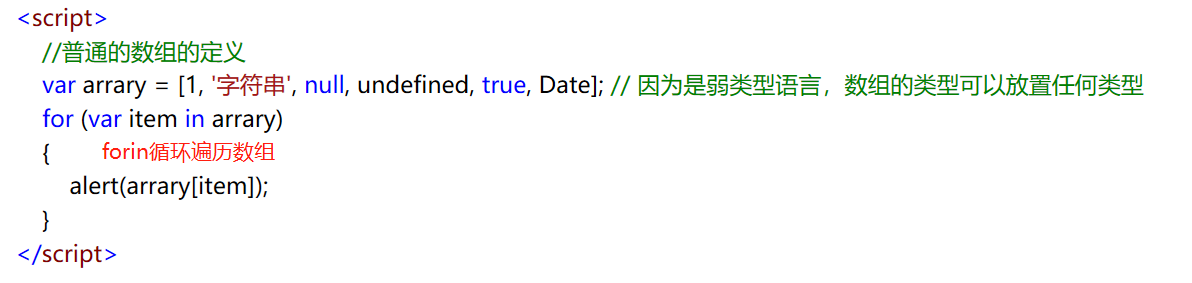
1. **JS中this所代表的含义**

-> 在JS中，this代表当前调用函数的对象

-> 比如说在类中使用this，那么this就代表的该类，如果直接在JS标签中使用this，this就代表的window对象，简而言之，谁调用了包含this的函数，那么this就代表谁

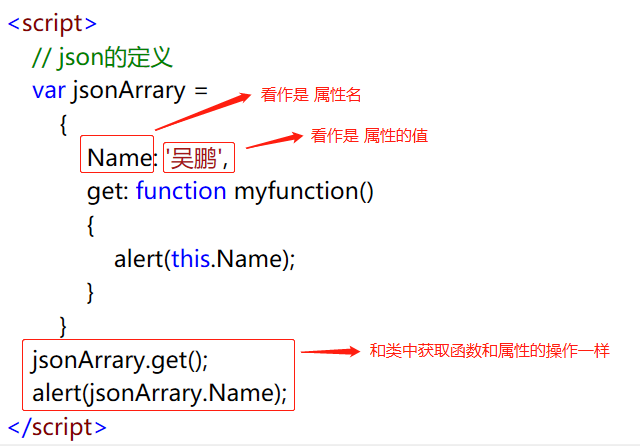


1. **JS中数组的定义**
2. 普通数组的定义
3. 因为JS是弱类型语言，所以数组的元素可以是任意类型

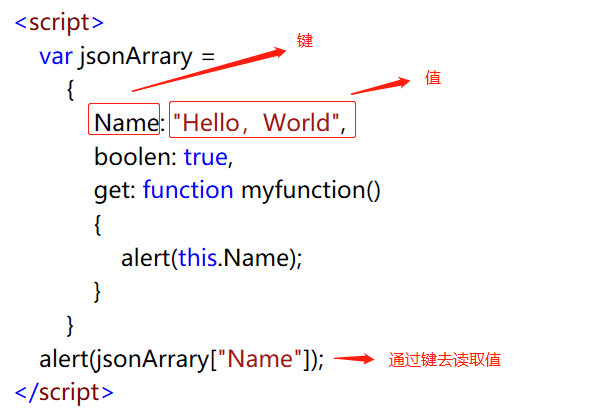


1. json（键值对数组）的定义
2. json（JavaScript Object Notation）：JS对象标记，可以看成是一种对象的简写形式，当声明了json的时候其实就已经实例化好了
3. 因为JS是弱类型语言，所以数组的元素可以是任意类型
4. 我们可以把这种数组看成是json也可以看成是一个键值对数组
5. 看作是json，通过属性名去读取属性的值

因为json是对象的一种简写形式，并且当我们声明完成json后也就实例化完成了json，所以，当我们把json赋值给一个变量的时候，我们就可以把变量名看作是json的实例名，所以我们可以通过：实例名.属性/函数()，去访问json的成员



1. 看作是键值对集合，通过键去读取值



1. **isNaN（Number）函数：能够判断一个变量的 值 是否不是数字（只要是值就可以，类型不定）**

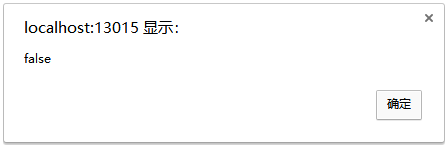
**参数类型**

**QQ截图20180414101652**

**代码规范**

QQ截图20180414101702

**实现效果**

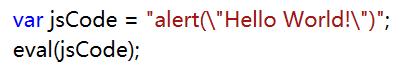


1. **eval（string x）：把一个字符串当作JavaScript代码来编译并执行**

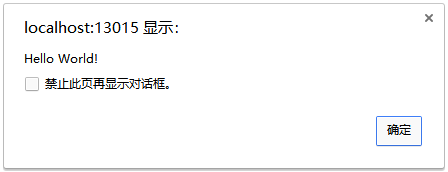
**参数类型**

**QQ截图20180414101925**

**代码规范**

****

**实现效果**

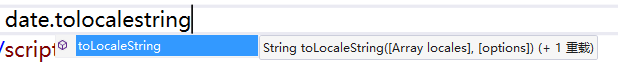


1. **Date对象：当我们创建了Date对象，就能够使用该对象中的一些函数来获取 浏览器所在的电脑的时间**

**创建Date对象**

**微信截图_20180409212715**

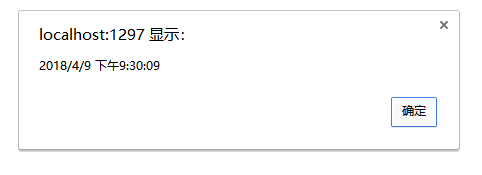
**Date对象中的某个函数（具体看W3CSchool）**

****

**代码规范**

****

**实现效果**

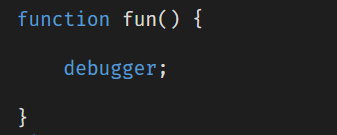


1. **JS中的字符串的函数（具体参照API文档）**



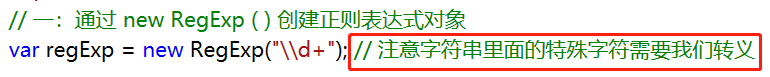
1. **debugger 关键字**

-> 使用JS进行Debug，除了手动的在浏览器调试窗口中手动设置断点外，我们还可以在JS的代码中手动添加该关键字，使网页在进入打开调试窗口的时候自动捕获至该关键字所声明处

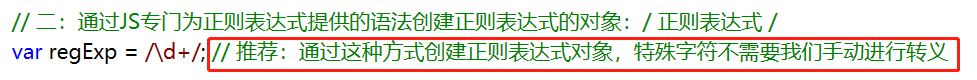


**JavaScript中使用正则表达式**

1. **创建正则表达式对象的两种方式（在JS中，只有使用正则表达式对象的一些函数才能获取正则匹配的结果）**
2. 通过 new RegExp( "正则表达式" ) 创建正则表达式对象



1. 通过JS专门为正则表达式提供的语法来创建正则表达式的对象：/ 正则表达式 / **推荐**



1. 全局模式与非全局模式的正则对象的构造方式

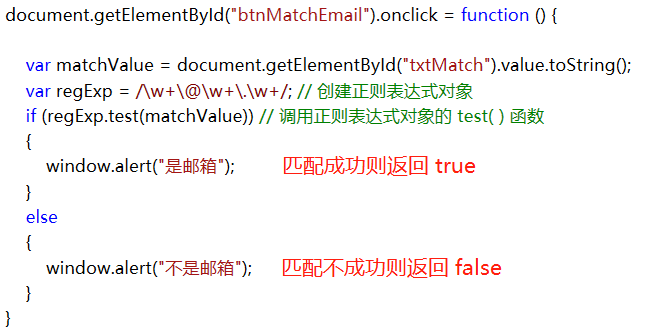
-> 全局模式：在构造正则表达式对象所需的正则表达式的结尾不加 g，则启用非全局模式

微信截图_20180525223602

-> 非全局模式：在构造正则表达式对象所需的正则表达式的结尾加 g，则启用全局模式

微信截图_20180525223614

1. **正则表达式对象的一些函数**
2. regExp.test ( string str ) ：使用正则表达式匹配字符串str中的值，匹配成功则返回 true，否则则返回 false



1. regExp.exec ( string str ) ：返回正则表达式匹配的结果，**需区分全局模式与非全局模式，不同的匹配模式，匹配的方式不同**

-> 无论是全局模式还是非全局模式，正则匹配**所返回的结果至始至终都是一个数组**，下标为0的元素则是当前匹配到的值的本身

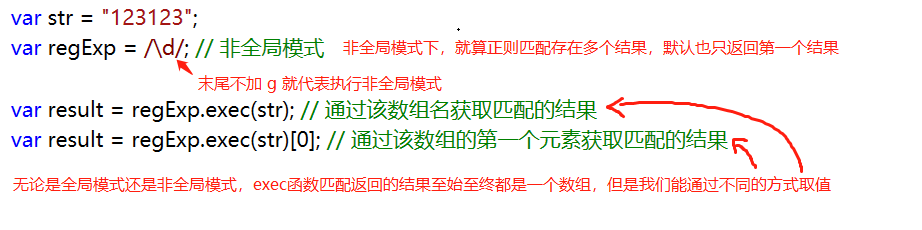
-> 如果我们在正则表达式中进行了分组匹配，那么下标为0的元素就是当前匹配到的值的本身，0后面的下标则是当前匹配到的值的分组后的结果

-> 我们除了能通过为0的下标访问当前匹配到的值的本身，也能直接通过该数组的名字访问到当前匹配到的值的本身

1. 非全局模式

-> 非全局模式下，就算正则匹配的结果存在多个值，至始至终也只返回第一个结果的值

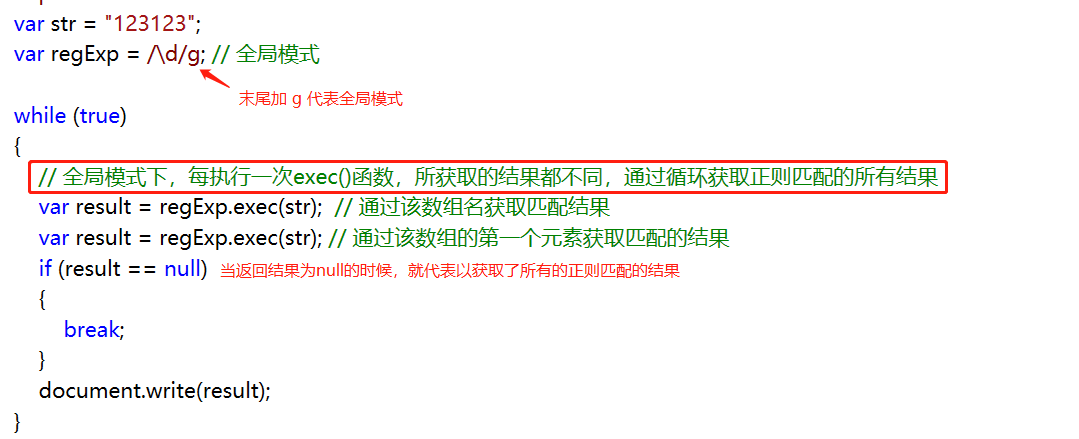
-> 当我们在构造正则表达式对象所需的正则表达式的结尾不加 g，则启用非全局模式



1. 全局模式

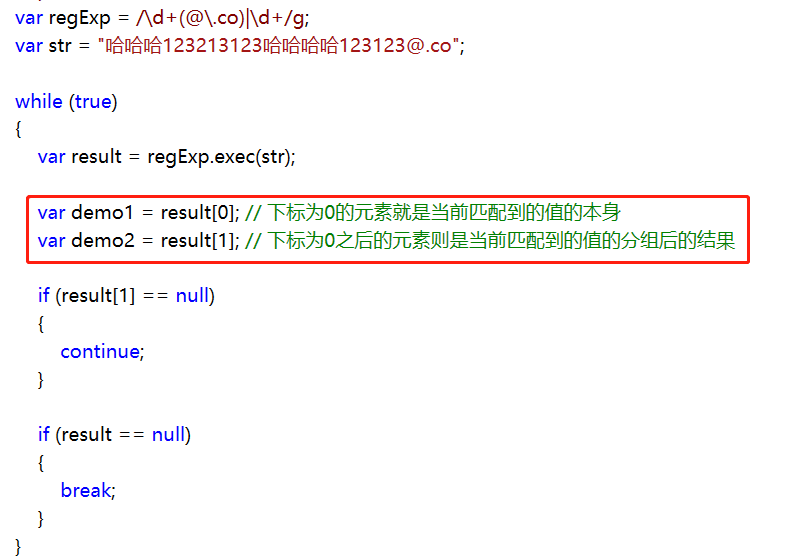
-> 全局模式下，我们需要通过循环执行 exec() 函数来获取正则匹配返回的所有结果，每执行一次exec() 函数则返回正则匹配的下一个结果，直至返回结果为null，则匹配结束

-> 当我们在构造正则表达式对象所需的正则表达式的结尾加 g，则启用全局模式



1. 分组匹配

-> 因为正则匹配所返回的结果至始至终都是一个数组，并且下标为0的元素就是当前匹配到的值的本身，所以，我们想访问当前匹配到的值的分组后的结果，必须从下标为1的元素开始获取



1. **string字符串对象的replace函数，指定正则对象为参数，则能够实现把正则匹配的结果替换成我们需要的值**

-> 注意正则对象全局模式与非全局模式下的使用，如果使用的是非全局模式，则正则表达式匹配所返回的结果至始至终都是第一个，所以，如果我们使用非全局模式下的正则对象进行replace，那么就只会替换掉第一个正则匹配结果的值，所以如果我们需要替换掉正则匹配结果所返回的全部的值，就需要使用全局模式

