# 匿名函数

当创建一个函数而没有指定变量引用时称函数为匿名函数，如

function（）{

console.log(“我是一个匿名函数”)；

}

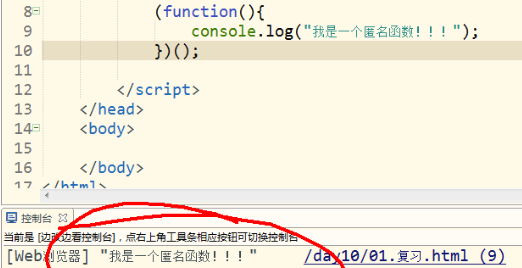
但这样会报错，因为function后面要跟函数名，而现在后面没有函数名，会把function和（）{}分开来解析，如果需要当成一个整体，需要一个（）括起来

（function（）{

console.log(“我是一个匿名函数”)；

}）

这样不会报错，但没有变量对它进行引用，结果会被当成垃圾回收。但这个匿名函数可以调用，整体就是一个函数对象，后面加一个（）就可以调用



这种函数称为立即执行函数，立即执行函数只会在创建完成后立即执行，并且只会执行一次，执行完后立即销毁

# 方法（method）

//创建一个对象

var obj = {name:"孙悟空",age:18};

//对象的属性也可以是对象

obj.brother = {name:"猪八戒",age:28};

console.log(obj.brother.name); //猪八戒

//对象的属性也可以是一个函数

obj.sayName = function(){

console.log("Hello 大家好 我是孙悟空~~~");

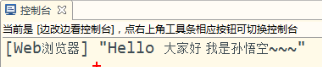
};

console.log(obj.sayName);

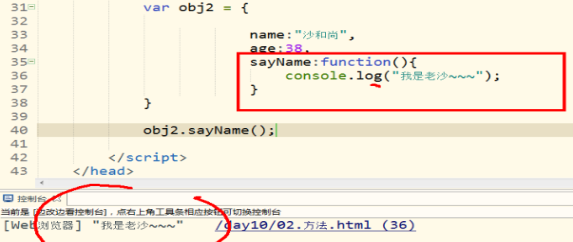


//调用函数

obj.sayName();



**当一个对象的属性是一个函数时，我们称这个函数是这个对象的方法，当我们调用对象中的函数时，我们称我们在调用对象的方法，函数和方法只是称呼上的不同，本质上没有区别。**



上图也是调用obj2的sayName方法

# 作用域（Scope）

作用域简单来说就是指一个变量的作用的范围

\* - 在JS中作用域分成两种：

\* 1.全局作用域

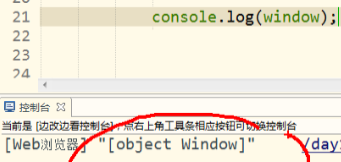
\* 2.函数作用域

全局作用域：

\* 1.所有直接在script标签中编写的代码都在全局作用域中

\* 2.全局作用域在打开网页时创建，在网页关闭时销毁

\* 3.全局作用域中有一个全局对象window，



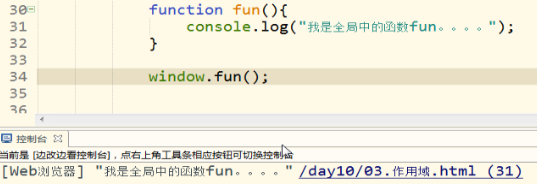
\* window代表的是浏览器的窗口，

\* 在全局作用域中创建的变量都会作为window对象的属性保存



console.log(window.a)和console.log(a)输出是一样的。而当一个变量比如c没有赋值时，console.log(c)会报错，输出c is not defined; console.log(window.c)不会报错，会输出undefined

\* 在全局作用域中创建的函数都会作为window对象的方法保存



调用fun函数使用window.fun()和使用fun()是一样的

\* 4.在全局作用域中创建的变量都是全局变量，可以在页面的任意位置访问

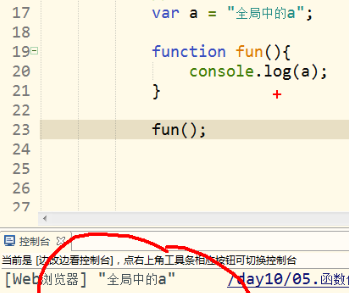
函数作用域

\* - 函数作用域可以理解为是全局中的小的作用域

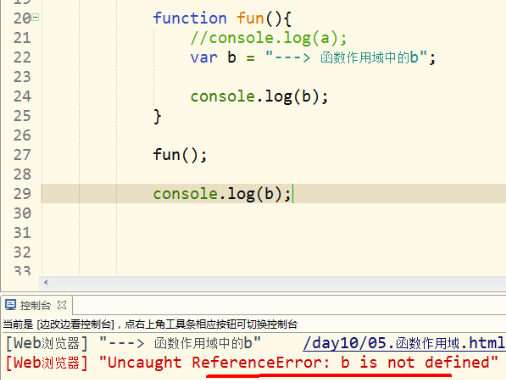
\* - 函数作用域在函数调用时创建（当没有调用函数时，函数中所有的内容均不会创建），在调用结束时销毁

\* 每调用一次函数就会创建一个新的函数作用域

\* - 在函数作用域中可以访问到全局变量,

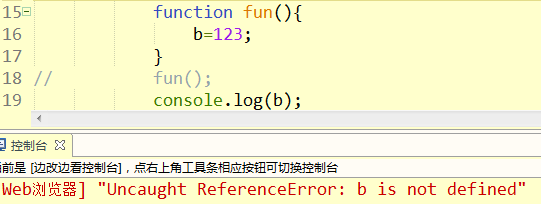


\* 而在全局中无法访问到函数作用域中的变量



\* 在函数中创建的变量如果不写var，则会变成全局变量

上图中，当fun函数中b前面不加var，b相当于全局变量，则可以输出，此时如果不调用函数fun，相当于函数作用域没有创建，函数中的内容不会创建，所以会报错。



\* - 当我们在函数中使用一个变量时，它会先在自身的作用域中寻找，

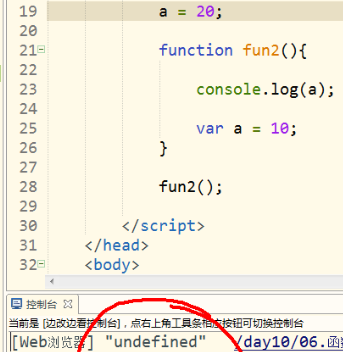
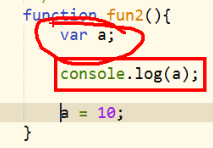
\* 如果有就直接使用，如果没有则去上一级作用域中寻找，

\* 找到则使用，没找到则继续寻找，直到找到全局作用域为止

\* 如果全局作用域中依然没有，则报错ReferenceError

\* - 在函数作用域中（不能在全局）也适用变量和函数的声明提前

在函数内部使用var关键字声明的变量也会被声明提前，它的声明会在函数作用域内所有的代码执行前执行

上面两个图fun函数中的内容是一样的，因为fun函数中有变量a，所以不会去全局中找a，只不过a没有值，所以返回undefined。而当a前面不写var后，函数作用域中没有变量a，会去全局找a，会返回全局中a的值20。

如果在函数作用域中想访问全局变量可以通过window对象来访问

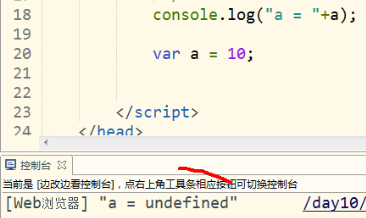


在函数内部使用函数声明创建的函数，也会在函数作用域内所有的代码执行之前被创建

# 声明提前

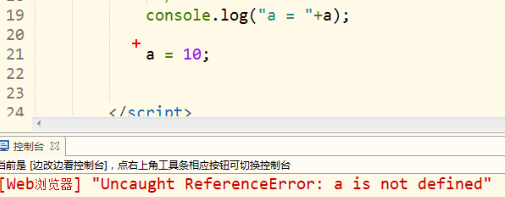
变量的声明提前

\* - 使用var关键字声明的变量，都会在所有的代码执行之前被声明，但是不会赋值



\* 赋值会直到执行到赋值的代码时才执行

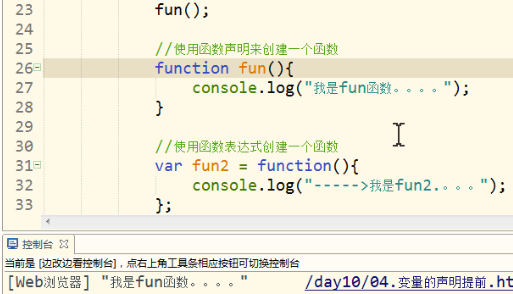
\* - 如果不使用var关键字声明变量，则不会有声明提前的特性



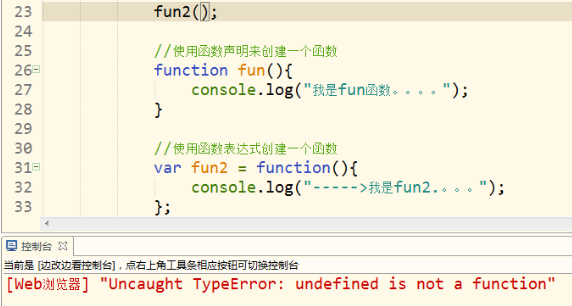
函数的声明提前

\* - 使用函数声明创建的函数，会在所有的代码执行之前创建，

\* 所以我们可以在函数声明前就去调用函数



\* - 使用函数表达式创建的函数，没有这个特性，所以不能在它创建之前调用



但上图中fun2由于使用了var，变量fun2会提前声明，但此时没有赋值，是undefined，undefined不能调用，所以报错。

# this

每次调用函数时，浏览器都会将一个对象作为隐藏的参数传递进函数

\* 这个对象就是函数执行的上下文对象，我们可以通过this来引用该对象

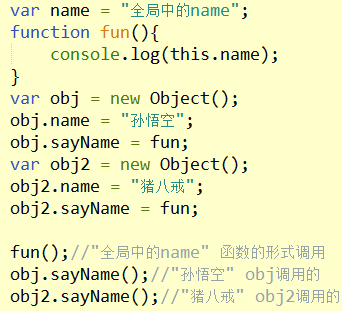
\* 根据函数的调用的形式不同，this的值也不同：

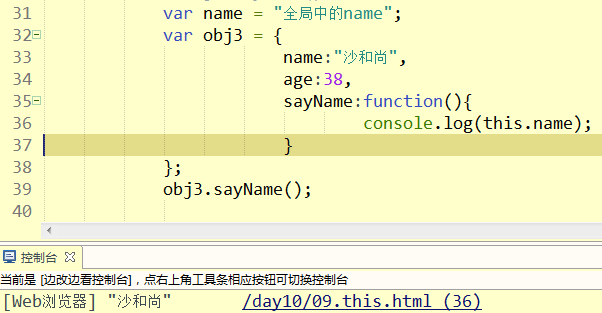
\* 1.以函数的形式调用时，this永远都是window

\* 2.以方法的形式调用时，谁调用this就是谁

其实当以函数的形式调用时，就相当于在调用函数前加window.

比如调用函数fun，fun（）就相当于window.fun（），本质上也是以方法的形式调用，一直是window调用，所以this永远都是window。



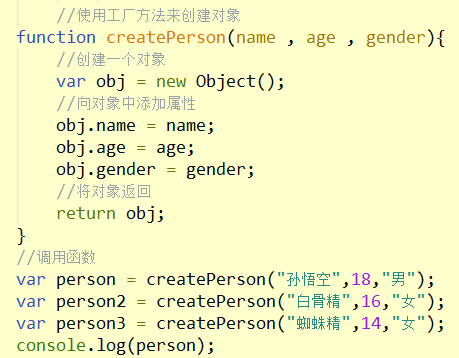


这个和以方法的形式调用是一样的



现在调的是obj3.sayName，也就是说调的function函数，而在function函数内部又声明了一个test，在test里输出this.name，然后又调了一个test（），也就是说调obj3.sayName，就相当于调test（），也就是说this.name输出是在test中输出的，this由test的调用形式决定，test是以函数的形式调用的，所以输出是window.name，即全局中的name。

# 工厂方法创建对象



上图中”调用函数”应该改为”创建对象”

上图中将返回值赋值给Person 是因为输出方便，如果不赋值，直接写console.log(createPerson(“孙悟空”,18,”男”));也可以，只不过麻烦。











# 构造函数（constructor）

构造函数是专门用来创建对象的函数

\* 构造函数就是一个普通的函数，不同的是构造函数需要通过new关键字来调用 // var p = new Person("孙悟空",18);

\* 构造函数一般都是首字母大写 // function Person(){};

\* 构造函数的执行的流程：

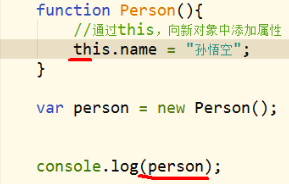
\* 1.创建一个新的对象

\* 2.将新创建的对象设置为函数中的this

\* 3.逐行执行函数

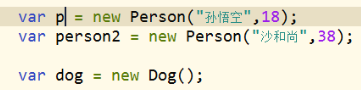
\* 4.将新建的对象作为返回值返回

\* 如果以构造函数的形式调用，this就是新创建的那个对象



\* 构造函数我们也可以称它为一个类，

\* 使用同一个构造函数创建的对象，我们称为一类对象



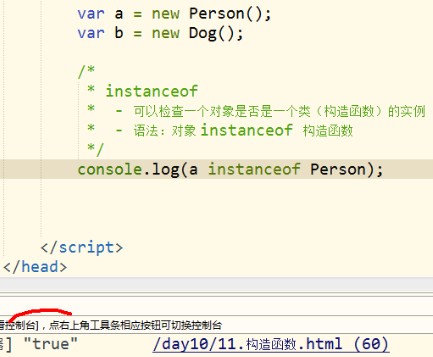
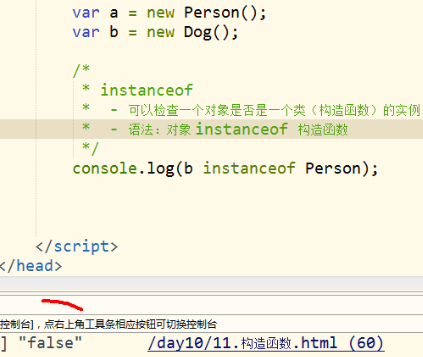
p和person2是一类对象，但和dog不是一类

\* 通过构造函数创建的对象，我们称这个对象是该类的实例

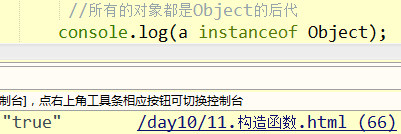
instanceof

\* - 可以检查一个对象是否是一个类（构造函数）的实例

\* - 语法：对象 instanceof 构造函数

所有的对象都是Object的实例



构造函数完善

//创建一个人类的构造函数

function Person(name , age){

//通过this，向新对象中添加属性

this.name = name;

this.age = age;

//向新的对象中添加一个sayName()方法

this.sayName = function(){

console.log(this.name);

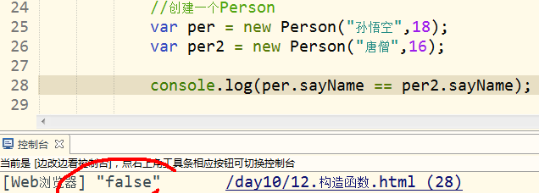
};

};

在构造函数中我们为每一个对象都添加了一个sayName()方法，

\* 而这个方法是在构造函数中创建的，构造函数每执行一次就要创建一个新的sayName()方法

\* 每个对象中的sayName()方法都是不同，有几个对象就有几个sayName()



\* 但是这些函数的功能又是一摸一样的，所以我们没有必要创造出这么多一摸一样的函数

可以将将函数定义到构造函数的外部

function fun(){

console.log(this.name);

};

function Person(name , age){

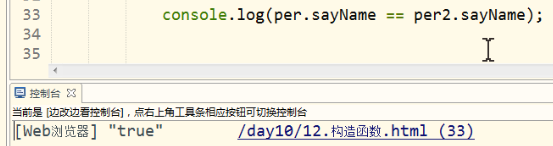
this.name = name;

this.age = age;

this.sayName = fun；

};

这样就只有一个函数，所有对象都在指向同一个fun函数，而且功能和上面的一样



将函数定义到全局作用域中 ，的确可以解决问题，但是将它定义到全局作用域将会导致全局作用域的命名空间被污染(定义一个函数后，以后再创建新函数就不能使用定义函数的名字)，同样也导致我们程序运行不安全(当定义函数以后又创建一个同样名字的函数，以前定义的函数就没用了)。这样就需要原形。

# 原形（prototype）

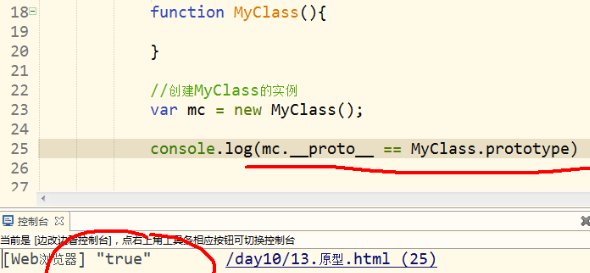
我们每次创建一个函数，浏览器都会为函数添加一个属性

\* 叫做prototype，这个属性对应的是一个对象

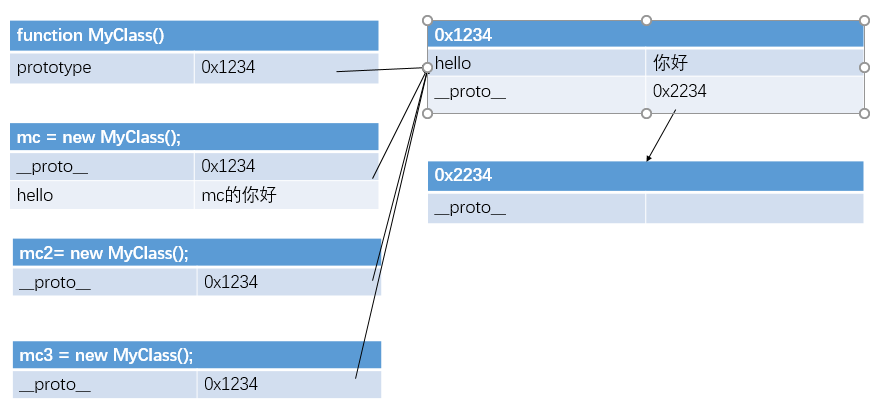
\* 这个对象就是我们说的原型对象。

\* - 如果仅仅以函数的形式去调用函数，则原型对象没有任何作用

\* - 当以构造函数的形式调用函数时，它所创建的对象中都会有一个隐含的属性，指向该函数的原型对象，我们可以通过\_\_proto\_\_来访问这个对象

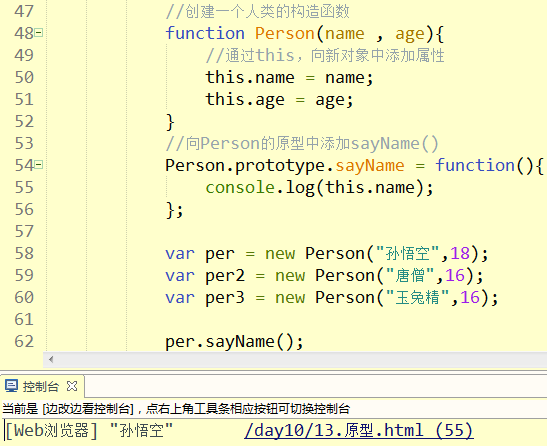


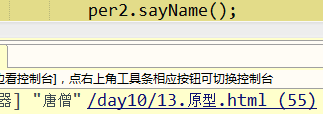
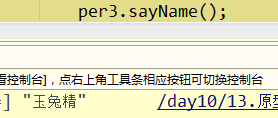
所有的同一类型的对象他们都共享同一个原型对象，这个原型对象就相当于一个公共的区域



\* - 我们可以将对象中共有的属性或方法统一添加到原型中，

\* 这样我们不用添加多个重复的属性或方法，也不会污染全局作用域



**实验**

function Person(name,age){

this.name=name;

this.age=age;

}

Person.prototype.sayName=function(){

console.log(this.name);

}

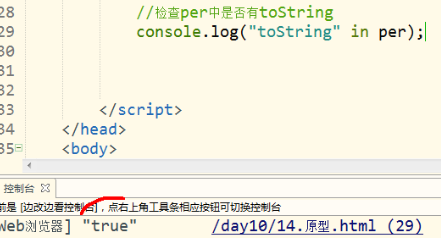
var per=new Person(“孙悟空”,18);

console.log(per); //[object Object]

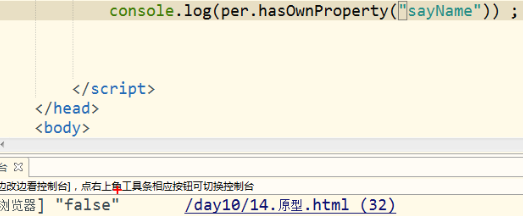
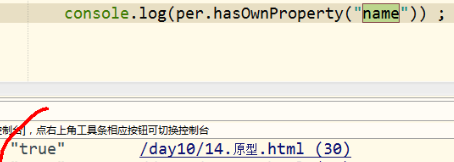
当我们打印一个对象时，控制台往往会输出 [object Object],此时实际上输出的内容是对象的toString()方法的返回值,也就是console.log(per.toString()); //[object Object]或者相当于var str=per.toString();

console.log(str); //[object Object],效果是一样的

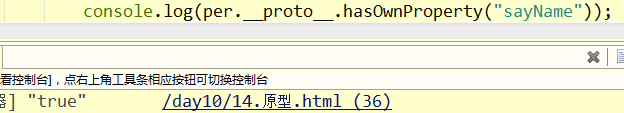
但Person中没有toString,检查per中是否有toString：



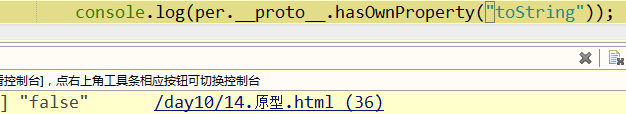
但是如果属性在对象的原型中，使用in也会返回true，可以使用hasOwnProperty来检查属性是否在对象中



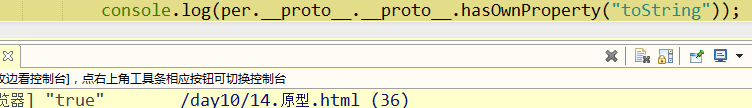
由于sayName是在Person的原形中，所以返回false，同样检查toString返回也是false，检查是否在原形中



证实sayName是在Person的原形中



证实toString不是在Person的原形中



证实toString是在Person的原形的原形中

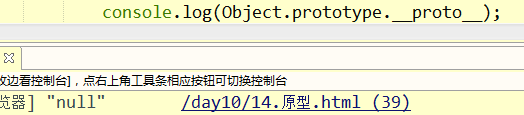
**结论**

\* - 当我们去调用一个对象的属性或方法时，它会先去对象自身中寻找,

\* 如果找到了则直接使用，如果没找到则去原型对象中寻找，如果原型中有则返回原型中的值

\* 如果原型中没有，则去原型的原型中寻找，找到了则直接使用，依次类推。。。

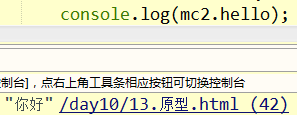
\* 注意：Object的原型的原型为null，所以会一直找到Object的原型，



(Object是一个函数，找它的原形使用prototype，而prototype是一

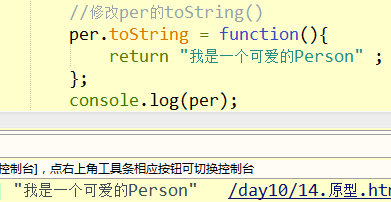
个对象，找它的原形使用\_\_proto\_\_)

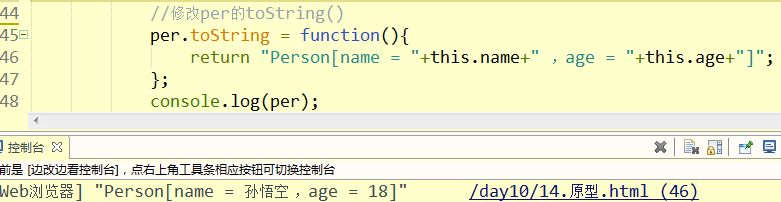
\* 如果他里面依然没有，则返回undefined



**延伸**

修改per的toString()





但此时创建一个新对象 var per2=new Person(“白骨精”，16)；

console.log(per2); //[object Object]

因为上面改的是Person的对象，而没有改Person的原形，上面的改法只对一个对象起作用，如果希望对Person所有的实例都起作用，需要改Person原形的toString

