# JavaScript高级

一、目标

理解面向对象开发思想，之前学习的基本是面向过程的编程。

掌握 JavaScript 面向对象开发相关模式。

掌握在 JavaScript 中使用正则表达式。

二、重新介绍 JavaScript

1、解释执行（javascript）：解释一行，执行一行，由浏览器负责解释成cpu能识别的，速度相对较慢，要来切换翻译、执行。它的好处在于灵活性-动态特性，随时可以给一个对象添加属性和方法。java等不行。

2、编译执行（java、c#）：它是一次性将所有的代码翻译成cpu能识别的01010，然后在一行一行去执行。

3、函数是 JavaScript 中的一等重要的，函数能做很多事情，能做参数，返回函数等等。在java中是类，函数是次重要的。

4、执行环境：在宿主环境（host environment）下运行，浏览器是最常见的 JavaScript 宿主环境。

在很多非浏览器环境中也使用 JavaScript ，例如 node.js

三、JavaScript 的组成

1、ECMAScript - 语法规范

- 变量、数据类型、类型转换、操作符

- 流程控制语句：判断、循环语句

- 数组、函数、作用域、预解析

- 对象、属性、方法、简单类型和复杂类型的区别

- 内置对象：Math、Date、Array，基本包装类型String、Number、Boolean

2、Web APIs

（1）BOM

- onload页面加载事件，window顶级对象，所有的内容都在window对象下

- 定时器

- location、history

（2）DOM

- 获取页面元素，注册事件

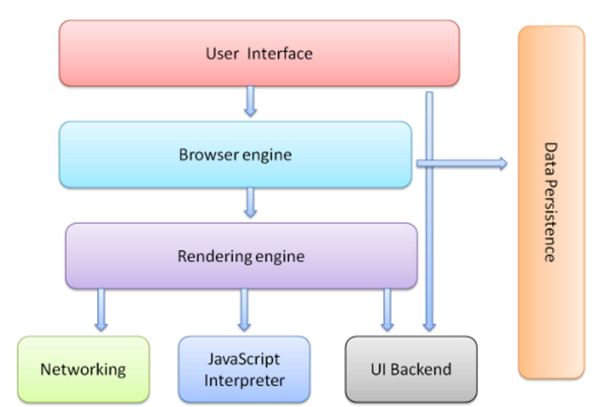
- 属性操作，样式操作

- 节点属性，节点层级

- 动态创建元素

- 事件：注册事件的方式、事件的三个阶段、事件对象

3、浏览器是如何工作的



此时先把域名由运营商的DNS（域名解析为IP地址）服务器转换成IP地址，

DNS再告诉浏览器这个IP地址，

浏览器再根据这个IP地址发送请求，告诉服务器，要去访问你的首页，

服务器把内容准备好之后，返回给浏览器。内容就是一些字符串，就是html文档内容，包涵html代码，css代码，js代码，然后浏览器中的部件去分别解析这些内容

（1）User Interface 用户界面，我们所看到的浏览器，包括工具栏，页面的可视区

（2）Browser engine 浏览器引擎，用来查询和操作渲染引擎

（3）\*Rendering engine 渲染引擎 用来显示请求的内容，负责解析HTML、CSS，并把解析的内容显示出来，并渲染成一棵DOM树，是在内存中存储的

（4）Networking 在浏览器输入www.baidu.com按下回车，它在网络上发送一个请求，负责发送请求和接收来自服务器返回的内容，接收到的基本是字符串

（5）\*JavaScript Interpreter(解析者) JavaScript解析器，负责解析、执行JavaScript的代码，操作内存中的DOM树，然后渲染引擎重新去更新这个DOM树，再告诉浏览器引擎已经更新完毕，把显示在用户界面

（6）UI Backend UI后端，用来绘制类似组合框和弹出窗口，如alert、

（7）Data Persistence(持久化) 数据持久化，数据存储 cookie、HTML5中的sessionStorage

4、JavaScript 执行过程

（1）JavaScript 运行分为两个阶段，不是真正的从上往下去执行：

（2）预解析

- 全局预解析（所有变量和函数声明都会提前；同名的函数和变量，函数的优先级高）

- 函数内部预解析（所有的变量、函数和形参都会参与预解析）

- 函数

- 形参

- 普通变量

（3） 执行

先预解析全局作用域，然后执行全局作用域中的代码，

在执行全局代码的过程中遇到函数调用就会先进行函数预解析，然后再执行函数内代码。

四、JavaScript 面向对象编程

1、面向对象介绍，什么是对象

（1）对象是单个事物的抽象。

一本书、一辆汽车、一个人（具体到个体上去，不能说是某类事物，如汽车是一类事情，并不是指某一辆）都可以是对象，一个数据库、一张网页、一个与远程服务器的连接也可以是对象。当实物被抽象成对象，实物之间的关系就变成了对象之间的关系，从而就可以模拟现实情况，针对对象进行编程。

（2）对象是一个容器，封装了属性（property）和方法（method）。

属性是对象的状态，方法是对象的行为（完成某种任务）。比如，我们可以把动物抽象为animal对象，使用“属性”记录具体是那一种动物，使用“方法”表示动物的某种行为（奔跑、捕猎、休息等等）。

在实际开发中，对象是一个抽象的概念，可以将其简单理解为：**数据集**或**功能集**。

ECMAScript-262 把对象定义为：无序属性的集合，其属性可以包含基本值、对象或者函数（**如果是函数一般称为方法**）。

严格来讲，这就相当于说对象是一组没有特定顺序的值。对象的每个属性或方法都有一个名字，而每个名字都映射到一个值。

提示：每个对象都是基于一个引用类型创建的，这些类型可以是系统内置的原生类型，也可以是开发人员自定义的类型。

2、什么是面向对象

面向对象不是新的东西，它只是过程式代码的一种高度封装，目的在于提高代码的开发效率和可维护性。

面向对象编程 —— Object Oriented Programming，简称 OOP ，是一种编程开发思想。它将真实世界各种复杂的关系，抽象为一个个对象，然后由对象之间的分工与合作，完成对真实世界的模拟。

在面向对象程序开发思想中，每一个对象都是功能中心，具有明确分工，可以完成接受信息、处理数据、发出信息等任务。因此，面向对象编程具有灵活、代码可复用、高度模块化等特点，容易维护和开发，比起由一系列函数或指令组成的传统的过程式编程（procedural programming），更适合多人合作的大型软件项目。

3、面向对象与面向过程：

面向过程就是亲力亲为，事无巨细，面面俱到，步步紧跟，有条不紊

面向对象就是找一个对象，指挥得结果

面向对象将执行者转变成指挥者

面向对象不是面向过程的替代，而是面向过程的封装

4、面向对象的特性：

（1）封装性

（2）继承性

（3）[多态性]抽象，多态性体现不明显，用抽象会更合适

5、程序中面向对象的基本体现

在 JavaScript 中，所有数据类型都可以视为对象，当然也可以自定义对象。自定义的对象数据类型就是面向对象中的类（ **Class ）的概念。ES6中有出现这个概念**

我们以一个例子来说明面向过程和面向对象在程序流程上的不同之处。

假设我们要处理学生的成绩表，为了表示一个学生的成绩，面向过程的程序可以用一个对象表示：

|  |
| --- |
| // 1、面向过程  var stu1 = {  name: 'a',  score: 100  };  var stu2 = {  name: 'b',  score: 80  }  console.log(stu1.name,stu1.score);  console.log(stu2.name,stu2.score);  // 2、面向对象  // 创建一个模板，用于创建对象（实例instance）  // 在javascript中创建对象的模板就是构造函数  // 在其他语言中创建对象的模板是类  function Student(name, score) {  this.name = name;  this.score = score;  this.print = function () {  console.log(this.name, this.score);  }  }  var stu1 = new Student('c',99);  var stu2 = new Student('d',88);  stu1.print();  stu2.print(); |

如果面向过程中出错，是比较难找的，要重新看下整个程序的流程，面向对象，只要看哪个对象出了问题，去找对应的构造函数就ok。

6、面向对象的设计思想

是从自然界中来的，因为在自然界中，类（Class）和实例（Instance）的概念是很自然的。

Class 是一种抽象概念，比如我们定义的 Class----------->Student ，是指学生这个概念，而实例（Instance）则是一个个具体的 Student ，比如， stu1 和 stu2 是两个具体的 Student。

所以，面向对象的设计思想是：

（1）抽象出 Class(构造函数)

（2）根据 Class(构造函数) 创建 Instance

（3）指挥 Instance 得结果

面向对象的抽象程度又比函数要高，因为一个 Class 既包含数据，又包含操作数据的方法。

遇到一个事情的时候，先分析哪些东西可以抽象成为对象

五、创建对象

1、简单方式

我们可以直接通过 new Object() 创建：

|  |
| --- |
| var car = new Object();  **car.name = '丰田';**  car.priace = 1000;  car.running = function () {  console.log('the car running');  } |

2、对象字面量 {} 可以看成是new Object()缩写来创建：

|  |
| --- |
| var car = {  **name:** '丰田',  priace: 1000,  running: function () {  console.log('the car.running');  }  } |

（1）对于上面的写法固然没有问题，但是假如我们要生成两个 person 实例对象呢？

|  |
| --- |
| var person1 = {  name: 'Jack',  age: 18,  sayName: function () {  console.log(this.name)  }  }  var person2 = {  name: 'Mike',  age: 16,  sayName: function () {  console.log(this.name)  }  } |

通过上面的代码我们不难看出，这样写的代码**太过冗余，重复性太高。**

3、工厂函数，解决代码重复问题

|  |
| --- |
| function createCar(name, priace) {  var car = new Object();  car.name = name;  car.priace = priace;  car.running = function () {  console.log('the car.running');  }  return car;  }  var car1 = createCar('本田',9999);  car1.running(); |

通过工厂模式封装，我们解决了创建多个相似对象代码冗余的问题，

但却没有解决对象识别的问题（即怎样知道一个对象的具体类型，只是知道是object，使用instance时候无法知道）。

4、构造函数

与 createCar() 有以下几点不同之处：

- 没有显示的创建对象

- 直接将属性和方法赋给了 this 对象

- 没有 return 语句

- 函数名使用的是大写的 Car

构造函数名字的第一个字母大写，用于区分普通函数

|  |
| --- |
| function Car(name, priace) {  this.name = name;  this.priace = priace;  this.running = function () {  console.log('the car running');  }  }  var car1 = new Car('保时捷', 99999);  console.log(car1 instanceof Car); // output: true  console.log(car1.constructor); |

（1）解析构造函数代码的执行

必须使用 new 操作符。以这种方式调用构造函数会经历以下 4 个步骤：

|  |
| --- |
| 1、会在内存中创建一个空的对象  2、设置构造函数中的this指针，指向刚刚创建好的对象  3、执行构造函数中的代码  4、返回刚刚创建的对象 |

（2）下面是具体的伪代码：

|  |
| --- |
| function Person (name, age) {  // 当使用 new 操作符调用 Person() 的时候，实际上这里会先创建一个对象  // var instance = {}  // 然后让内部的 this 指向 instance 对象  // this = instance  // 接下来所有针对 this 的操作实际上操作的就是 instance  this.name = name  this.age = age  this.sayName = function () {  console.log(this.name)  }  // 在函数的结尾处会将 this 返回，也就是 instance  // return this  } |

（3）构造函数和实例对象的关系

使用构造函数的好处不仅仅在于代码的简洁性，更重要的是我们可以识别对象的具体类型了。

在每一个实例对象中同时有一个 constructor 属性（**并不是直接有这个属性，见构造函数、实例、原型三者之间的关系**），该属性指向创建该实例的构造函数： **constructor 是实例的属性的说法不严谨，之所以能直接访问**



对象的 constructor 属性最初是用来标识对象类型的，但是，如果要检测对象的类型，还是使用 instanceof 操作符更可靠一些。

|  |
| --- |
| console.log(car1.constructor === Car);  var array = [];  console.log(**array.constructor === Array**); |

无法使用typeof获取对象的具体类型，返回的都是object。

获取对象的具体类型，对象有一个constructor属性，不建议使用，它是可以被修改的。

**instanceof 判断某个对象是否是某个构造函数的实例（对象）。**

（4）总结：

构造函数是根据具体的事物抽象出来的抽象模板。

实例对象是根据抽象的构造函数模板得到的具体实例对。

每一个实例对象都具有一个 **constructor 属性，指向创建该实例的构造函数**。

注意： **constructor 是实例的属性的说法不严谨，之所以能直接访问是系统通过\_\_proto\_\_指针指向的原型对象去查找的，原型链相关之知识**。

可以通过实例的 constructor 属性判断实例和构造函数之间的关系。

注意：这种方式不严谨，推荐使用 instanceof 操作符，后面学原型会解释为什么。

5、构造函数的问题

使用构造函数带来的最大的好处就是创建对象更方便了，但是其本身也存在一个**浪费内存的问题**：

|  |
| --- |
| function Student(name, age, sex) {  this.name = name;  this.age = age;  this.sex = sex;  this.sayHi = function () {  console.log('hello world');  }  }  var stu1 = new Student('a', 15, '男');  var stu2 = new Student('b', 33, '女');  console.log(stu1.sayHi === stu2.sayHi);**// => false，它比较的其实是两个函数的地址，它们的内存地址是不一样的** |

在该示例中，从表面上好像没什么问题，但是实际上这样做，有一个很大的弊端。

那就是对于每一个实例对象，sayHi 都是一模一样的内容。

每一次生成一个实例，都必须为重复的内容，多占用一些内存，如果实例对象很多，会造成极大的内存浪费。

（1）对于这种问题我们可以把需要共享的函数定义到构造函数外部：

|  |
| --- |
| function Student(name, age, sex) {  this.name = name;  this.age = age;  this.sex = sex;  this.sayHi = sayHi;  }  function sayHi() {  console.log('hello world ' + this.name);  // 在书写函数的时候，this的指向是无法确认的，只有当它**被调用**，执行的时候，才能确定，书写或者注册的时候暂时无法确定的。  }  var stu1 = new Student('a', 15, '男');  var stu2 = new Student('b', 33, '女');  stu1.sayHi(); |

这样确实可以了，但是如果有多个需要共享的函数的话就会造成全局命名空间冲突的问题。

（2）可以把多个函数放到一个对象中用来避免全局命名空间冲突的问题：

|  |
| --- |
| function Student(name, age, sex) {  this.name = name;  this.age = age;  this.sex = sex;  this.sayHi = fns.sayHi;  }  var fns = {  sayHi: function () {  console.log('hello world');  }  } |

至此，我们利用自己的方式**基本上解决了构造函数的内存浪费问题**。

但是代码看起来还是那么的格格不入，那有没有更好的方式呢？

6、静态成员跟实例成员

1. 静态成员

|  |
| --- |
| 直接通过对象来调用的属性和方法，称为静态成员（一般是字面量方式）。**没有动态成员**  工具中使用静态成员，如Math，可以求出最大值，最小值等等，这个Math工具只有一个。  var car = {  name: '丰田',  priace: 1000,  running: function () {  console.log('the car.running');  }  }  **car.running(); // 静态成员** |

（2）实例成员

|  |
| --- |
| 实例是对象的另外一种说法。当有很多个对象的时候，使用构造函数来创建。  实例成员：构造函数中的成员就是实例成员，它的调用需要创建出实例之后，才能通过实例去调用。  function Car(name, priace) {  this.name = name;  this.priace = priace;  this.running = function () {  console.log('the car running');  }  }  var car1 = new Car('保时捷', 99999);  car1.running(); |

六、原型

1. 更好的解决方案： prototype

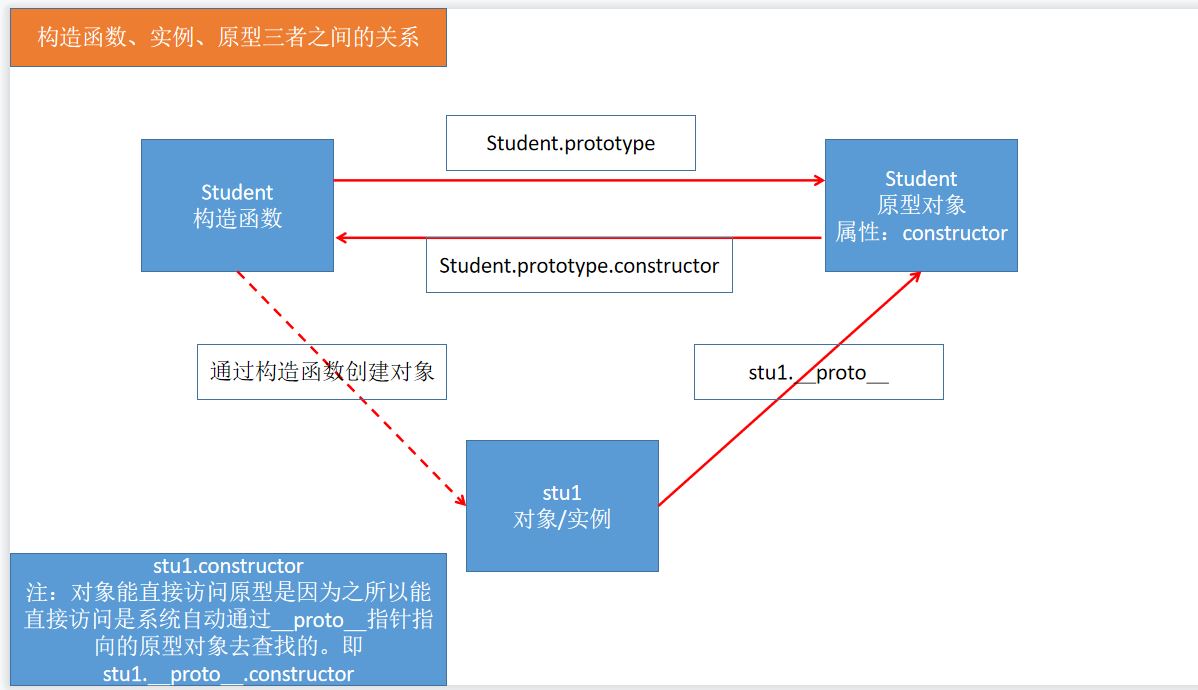
JavaScript 规定，**每一个构造函数（不是实例，容易混淆）都有一个 prototype（原型 / 原型对象） 属性**，它其实是一个对象。这个对象的所有属性和方法，都会被构造函数的所拥有。

这也就意味着，我们可以把所有对象实例需要共享的属性和方法直接定义在 prototype 对象上。

|  |
| --- |
| function Student(name, age, sex) {  this.name = name;  this.age = age;  this.sex = sex;  }  **Student.prototype.sayHi** = function () {  console.log('hello world');  }  **Student.prototype.class** = '三班';  var stu1 = new Student('a', 15, '男');  var stu2 = new Student('b', 33, '女');  stu1.sayHi();// **实例对象可以直接访问原型对象成员。之所以能直接访问是系统通过\_\_proto\_\_指针指向的原型对象去查找的，原型链相关之知识**  console.log(stu1.sayHi === stu2.sayHi); // output true  console.log(Student.prototype);  通过该构造函数创建的对象，可以直接访问构造函数.prototype中的成员（属性、方法），**之所以能直接访问是系统通过\_\_proto\_\_指针指向的原型对象去查找的，原型链相关之知识** |
| 1 |

**Student.prototype.constructor指向了构造函数本身**

2、构造函数、实例、原型三者之间的关系



（1）任何函数都具有一个 prototype 属性，该属性是一个对象。

|  |
| --- |
| function fn() {  console.log('f');  }  console.log(fn.prototype.constructor);  1 |

（2）构造函数的 prototype 对象默认都有一个 constructor 属性，指向 prototype 对象所在函数。

（3）通过构造函数得到的实例对象内部会包含一个指向构造函数的 prototype 对象的指针 **\_\_proto\_\_**。（ \_\_proto\_\_ 是非标准属性。）

|  |
| --- |
| console.log(stu1.constructor);  console.log(stu1.\_\_proto\_\_.constructor);  二者输出的结果是一致的 |

（4）实例对象可以**直接访问原型对象成员**。**之所以能直接访问是系统通过\_\_proto\_\_指针指向的原型对象去查找的，原型链相关之知识**

console.log(stu1.constructor);

（5）总结：

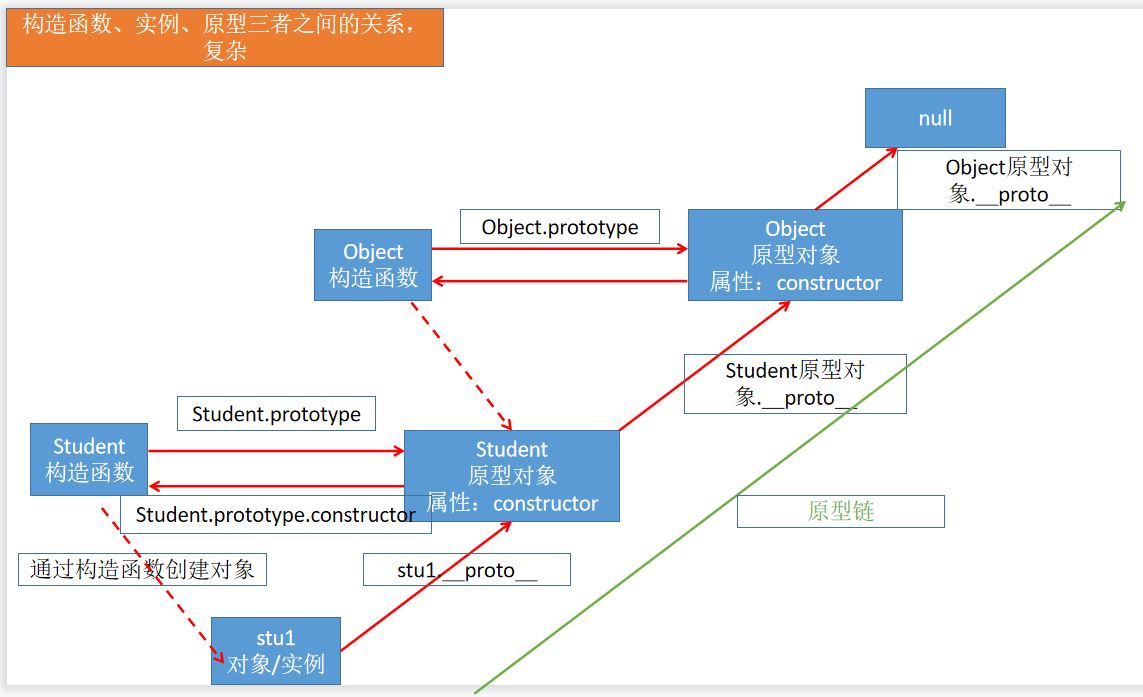
- 任何函数都具有一个 prototype 属性，该属性是一个对象。

- 构造函数的 prototype 对象默认都有一个 constructor 属性，指向 prototype 对象所在函数。

- 通过构造函数得到的实例对象内部会包含一个指向构造函数的 prototype 对象的指针 \_\_proto\_\_。

- 所有实例都直接或间接**继承了**原型对象的成员。

3、属性成员的搜索原则：原型链



构造函数-实例-原型对象 三者之间的关系后，接下来我们来解释一下

（1）为什么实例对象可以**访问原型对象中的成员**。

每当代码读取某个对象的某个属性时，都会执行一次搜索，目标是具有给定名字的属性

|  |
| --- |
| 1、搜索首先从对象实例本身开始  2、如果在实例中找到了具有给定名字的属性，则返回该属性的值  3、如果没有找到，则继续搜索**指针（\_\_proto\_\_）**指向的原型对象，在原型对象中查找具有给定名字的属性  4、如果在原型对象中找到了这个属性，则返回该属性的值 |
| <script type="text/javascript">  function Student(name, age) {  this.name = name;  this.age = age;  this.sayHi = function () {  console.log('hello');  }  }  Student.prototype.sayHi = function () {  console.log('Hi' + this.name);  }  var stu1 = new Student('Mike',22);  stu1.sayHi();  </script>  **结果输出的是 hello** |
| 也就是说，在我们调用 stu1.sayHi() 的时候，会先后执行两次搜索：  - 首先，解析器会问：“实例 stu1有 sayHi 属性吗？”答：“没有。  - ”然后，它继续搜索，再问：“ stu1的原型有 sayHi 属性吗？”答：“有。  - ”于是，它就读取那个保存在原型对象中的函数。  - 当我们调用 stu2.sayHi() 时，将会重现相同的搜索过程，得到相同的结果。 |

而这正是多个对象实例共享原型所保存的属性和方法的基本原理。

（2）toString()函数的查找

所有的对象都有toString的方法，根据原型链查找规则，最终查找到Object这个原型对象的成员

console.log(stu1.toString());自己创建的对象返回的是固定的形式[object Object]，object代表的是一个对象，Object代表的是类型，它获取的不是具体的类型，而是所有对象的祖宗。

|  |
| --- |
| function Student(name, age) {  this.name = name;  this.age = age;  }  Student.prototype.sayHi = function () {  console.log('Hi' + this.name);  }  var stu1 = new Student('Mike',22);    console.log(stu1.toString());  console.log(stu1.\_\_proto\_\_);  console.log(stu1.\_\_proto\_\_.\_\_proto\_\_);  1 |

（3）设置属性

要设置的属性如果在原型对象上，在设置值的时候，是不会去搜索原型链的，而是直接给对象新增这个属性。

|  |
| --- |
| function Student(name, age) {  this.name = name;  this.age = age;  }  Student.prototype.sayHi = function () {  console.log('Hi' + this.name);  }  ***Student.prototype.test = 'test';***  var stu1 = new Student('Mike',22);  var stu2 = new Student('Jon',25);  ***stu1.test = '111';***  console.log(stu1.test);  console.log(stu2.test);  1 |

（4）读取属性

读取属性，如果在对象本身查找属性，如果没有找到的话，就会去原型链上查找。

（5）总结：

先在自己身上找，找到即返回。

自己身上找不到，则沿着原型链向上查找，找到即返回。

如果一直到原型链的末端还没有找到，则返回 undefined。

4、注意点

一般情况下，对象的属性在构造函数中来设置。对象的方法在构造函数的原型对象中来设置。

（1）前面例子中每添加一个属性和方法就要敲一遍 Person.prototype 。

为减少不必要的输入，更常见的做法是用一个包含所有属性和方法的对象字面量来重写整个原型对象：

|  |
| --- |
| function Student(name, age) {  this.name = name;  this.age = age;  }  /\*Student.prototype.sayHi = function () {  console.log('Hi' + this.name);  }  Student.prototype.eat = function () {  console.log('eating');  }\*/  **Student.prototype = {**  **sayHi: function () {**  **console.log('Hi' + this.name);**  **},**  **eat: function () {**  **console.log('eating');**  **}**  **}**  var stu1 = new Student('Mike',22);  console.log(stu1.constructor);  1  结果输出的是 f Object() 系统的对象构造函数。  原因是：我们给Student.prototype属性设置了一个新的对象，这个对象里边并没有construtor属性，所以在原型链上查找，最后在原型对象的原型对象上找到了constructor，就返回了Object。 |

（2）手动将 constructor 指向正确的构造函数，保持 constructor 的指向正确

|  |
| --- |
| Student.prototype = {  **constructor: Student,**  sayHi: function () {  console.log('Hi' + this.name);  },  eat: function () {  console.log('eating');  }  } |

（3）想使用的新的prototype 的方法，必须在其创建完之后，才能使用

|  |
| --- |
| function Student(name, age) {  this.name = name;  this.age = age;  }  **var test = new Student('MM', 20);**  **test.sayHi();**  Student.prototype = {  constructor: Student,  sayHi: function () {  console.log('Hi ' + this.name);  },  eat: function () {  console.log('eating');  }  }  var stu1 = new Student('Mike',22);  stu1.sayHi();  这里会出错，因为此时在原型链上找不到sayHi方法 |

5、内置对象方法拓展

|  |
| --- |
| <script type="text/javascript">  var array = [1,2,3,4,5];  Array.prototype.getSum = function () {  var sum = 0;  console.log(**this**); // 在调用的时候会输出 array，没有调用之前是不知道指向谁的  for (var i = 0, len = this.length; i < len; i++) {  sum += this[i];  }  return sum;  }  /\*Array.prototype = { // 这样写直接报错  getSum: function () {  var sum = 0;  for (var i = 0, len = this.length; i < len; i++) {  sum += this[i];  }  return sum;  }  }\*/  console.log(array.getSum());// output 15  </script> |

Array、Sring等内置对象的prototype属性，**系统不允许修改属性的值**。可以通过给它添加新的方法。

1. 小案例：随机方块

工具对象，目前只有一个，所有使用字面量方式，可以不用构造函数。

引入js文件的顺序也是有讲究的，先引入需要被调入的js文件

