# 面向对象进阶知识

1. 面向对象的原型prototype
   1. Javascript 规定，每一个构造函数都有一个 prototype 属性，指向另一个对象。
   2. 这个对象的所有属性和方法，都会被构造函数的实例继承。
   3. 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 Person (name, age) {

this.name = name

this.age = age

}

console.log(Person.prototype)

Person.prototype.type = 'human'

Person.prototype.sayName = function () {

console.log(this.name)

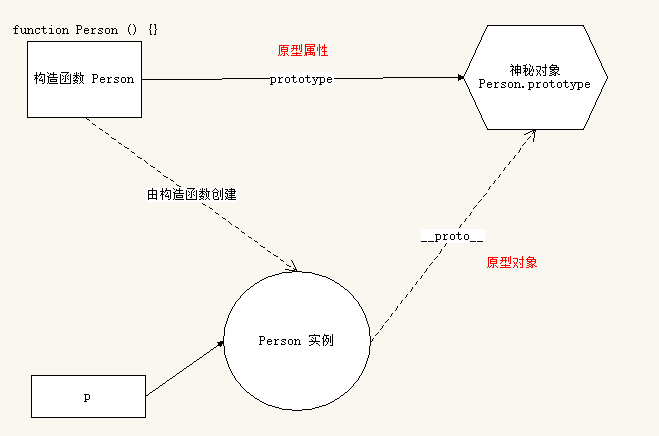
}

var p1 = new Person(...)

var p2 = new Person(...)

console.log(p1.sayName === p2.sayName) // => true

* 1. 这时所有实例的 type 属性和 sayName() 方法，其实都是同一个内存地址，指向 prototype 对象，因此就提高了运行效率。



任何函数都具有一个 prototype 属性，该属性是一个对象。

function F () {}

console.log(F.prototype) // => object

F.prototype.sayHi = function () {

console.log('hi!')

}

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。

console.log(F.constructor === F) // => true

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\_\_。

var instance = new F()

console.log(instance.\_\_proto\_\_ === F.prototype) // => true

\_\_proto\_\_ 是非标准属性。

实例对象可以直接访问原型对象成员。

instance.sayHi() // => hi!

**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

1. 属性成员的搜索原则：原型链

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

- 搜索首先从对象实例本身开始

- 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

- 如果没有找到，则继续搜索指针指向的原型对象，在原型对象中查找具有给定名字的属性

- 如果在原型对象中找到了这个属性，则返回该属性的值

也就是说，在我们调用 person1.sayName() 的时候，会先后执行两次搜索：

- 首先，解析器会问：“实例 person1 有 sayName 属性吗？”答：“没有。

- ”然后，它继续搜索，再问：“ person1 的原型有 sayName 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 person2.sayName() 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。

**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

我们注意到，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 Person (name, age) {

this.name = name

this.age = age

}

Person.prototype = {

type: 'human',

sayHello: function () {

console.log('我叫' + this.name + '，我今年' + this.age + '岁了')

}

}

在该示例中，我们将 Person.prototype 重置到了一个新的对象。

这样做的好处就是为 Person.prototype 添加成员简单了，但是也会带来一个问题，那就是原型对象丢失了 constructor 成员。

所以，我们为了保持 constructor 的指向正确，建议的写法是：

function Person (name, age) {

this.name = name

this.age = age

}

Person.prototype = {

constructor: Person, // => 手动将 constructor 指向正确的构造函数

type: 'human',

sayHello: function () {

console.log('我叫' + this.name + '，我今年' + this.age + '岁了')

}

}

#### **原生对象的原型**

**每一个内置的对象的构造函数 都有原型，几乎大部分的方法都挂载到了原型身上**

- Object.prototype

- Function.prototype

- Array.prototype

- String.prototype

- Number.prototype

- Date.prototype

练习：为数组对象和字符串对象扩展原型方法。

Array.prototype.push = function(){

For(var i = 0; i< arguments.length; i++) {

This[this.length] = arguments[i];

}

}

Array.prototype.pop = function(){

This.length = this.length -1;

}

Array.prototype.slice = function(n,m) {

Var arr = [];

For (var i = n; i< m; i++){

arr[arr.length] = this[i]

}

Return arr;

}

Array.prototype.norepeat = function(){

Var arr = [];

For(var i =0; i<this.length;i++){

var onOff = true;

For(var j = 0; j< arr.length; j++) {

If(arr[j] == this[i]) {

onOff = false;

Break;

}

}

If(onOff){

arr[arr.length] = this[i];

}

}

Return arr;

}

String.prototype.indexOf = function(str,idx){

Idx = idx || 0;

For(var i = idx; i< this.length;i++) {

If(str == this[i]) {

Return i;

}

}

Return -1;

}

String.prototype.trim = function() {

Return this.replace(/^\s+|\s+$/g,””);

}

最好不要轻易修改实例之间互相共享这些数组或者对象成员，一旦修改的话会导致数据的走向很不明确而且难以维护。

尤其是系统内置的对象的构造函数！

原型的使用建议：

- 私有成员（一般就是非函数成员）放到构造函数中

- 共享成员（一般就是函数）放到原型对象中

- 如果重置了 prototype 记得修正 constructor 的指向

**案例：**

拖拽面向对象的案例封装（面向对象tab切换）