面向对象这块知识点，是javascript中的重点与难点。许多库的实现都是基于面向对象思想，而且平常面试中，也是一个高频考点甚至是必考点。

很多文章一开头就各种介绍原型链、原型、构造函数等等，这样容易让一些javascript初学者感到混乱，因此本文采取渐进式的讲解，尽量简单易懂的描述来展示面向对象的相关知识，结合图文与例子来帮助大家快速理解面向对象。

## 对象的定义

ECMA-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。”严格来讲， 这就相当于说对象是一组没有特定顺序的值。简单的说，对象是由一些无顺序的key-value对组成的，其中value可以包含基本值，对象或者函数。

比如：

var person = {

name: 'Tom',

age: 18,

getName: function() {},

parent: {}

}

在该例子中，person就是一个对象。

**创建对象**

在javascript中，创建对象无非就两种形式：

1. new的方式创建

var obj = new Object();

2.对象字面量的形式创建一个简单的对象

var obj = {};

本质上，对象都是通过函数创建的，第2中创建对象的方式是第一种的语法糖而已。

**给对象添加方法**

给对象添加方法有两种方式：

// 第一种

var person = {};

person.name = "TOM";

person.getName = function() {

return this.name;

}

// 第二种

var person = {

name: "TOM",

getName: function() {

return this.name;

}

}

**对象属性和方法的访问**

比如有以下对象：

var student = {

name: 'TOM',

}

那么我们访问对象student的方式有两种：

1. student .name
2. student [‘name’]

这两种方式的区别是，第二种可以用于属性名为变量时候的访问，比如

var abc = ‘name’

student [abc] // 等同于abc

## 工厂模式-提升创建对象效率

虽然使用上述方式创建对象比较简单，但是如果我们需要100个student对象时，且100个student对象只是name不同，就不得不定义100个student对象，但只是name不同。这种方式会造成太多相似对象，代码变得十分臃肿。

工厂模式是软件工程领域一种广为人知的设计模式，这种模式抽象了创建具体对象的过程。提供一个对象的模子，像工厂一样，需要多少个对象就复制出多少个。

看代码：

function createPerson(name, age){

var o = new Object();

o.name = name;

o.age = age;

o.sayName = function(){

return this.name

};

return o;

}

var person1 = createPerson("Nicholas", 29);

var person2 = createPerson("Greg", 27);

createPerson内部实现了一个模子对象o，并支持函数入参作为模子对象o的一些属性，通过调用createPerson函数和传入对应参数，就可以得到对应的实例对象。一样的key，定制化的value。

工厂模式虽然解决了创建 多个相似对象的问题，但却没有解决对象识别的问题（即怎样知道一个对象的类型）。

var obj = {};

var foo = function() {}

console.log(obj instanceof Object); // true

console.log(foo instanceof Function); // true

随着 JavaScript 的发展，又一个新模式出现（ps：有同学留下学不动的泪水吗）

## 构造函数

构造函数模式的目的就是为了创建一个自定义类，并且创建这个类的实例。构造函数模式中拥有了类和实例的概念，并且实例和实例之间是相互独立的，简单的说就是不同的对象。

构造函数就是一个普通的函数，创建方式和普通函数没有区别，但是有两点与普通函数不同：

1. 构造函数习惯上首字母大写。
2. 就是调用方式的不同，普通函数是直接调用，而构造函数需要使用new关键字来调用。

由于new关键字调用，构造函数与普通函数的表现是不同。

function demo() {

console.log(this);

}

demo(); // window

new demo(); // demo

为了方便演示，我们没有使用Demo大写模式。

通过构造函数，我们就能够判断实例与对象关系了。

var Person = function(name, age) {

this.name = name;

this.age = age;

this.getName = function() {

return this.name;

}

}

var p1 = new Person('Ness', 20);

console.log(p1.getName()); // Ness

console.log(p1 instanceof Person); // true

通过上述两个例子，你是不是有些疑问了呢？

为什么会出现这种调用差异呢？new关键字到底做了什么呢？构造函数是怎么判断实例与对象关系的？直接手撕代码实现下new过程，一探究竟：

// 先一本正经的创建一个构造函数，其实该函数与普通函数并无区别

var student= function(name, age) {

this.name = name;

this.age = age;

this.getName = function() {

return this.name;

}

}

// 将构造函数以参数形式传入

function New(func) {

// 声明一个中间对象，该对象为最终返回的实例

var res = {};

if (func.prototype !== null) {

// 将实例的原型指向构造函数的原型

res.\_\_proto\_\_ = func.prototype;

}

// ret为构造函数执行的结果，这里通过apply，将构造函数内部的this指向修改为指向res，即为实例对象

var ret = func.apply(res, Array.prototype.slice.call(arguments, 1));

// 当我们在构造函数中明确指定了返回对象时，那么new的执行结果就是该返回对象

if ((typeof ret === "object" || typeof ret === "function") && ret !== null) {

return ret;

}

// 如果没有明确指定返回对象，则默认返回res，这个res就是实例对象

return res;

}

// 通过new声明创建实例，这里的p1，实际接收的正是new中返回的res

var p1 = New(student, 'tom', 20);

console.log(p1.getName());

// 当然，这里也可以判断出实例的类型了

console.log(p1 instanceof Person); // true

为了更进一步理解，我们简单的总结下new过程：

1. 声明创建一个新的中间对象
2. 把构造函数的原型对象赋值给中间对象的原型，即中间对象的原型指向构造函数的原型
3. 将构造函数的this指向中间对象
4. 把中间对象作为实例对象返回

## 原型

看代码：

function Person (name, age) {

this.name = name;

this.age = age;

this.getName = function() {

return this.name;

}

}

var p1 = new Person('Ness', 20);

虽然构造函数解决了实例类型判断问题，但是如上述例子，如果我们要创建100个Person实例就会有一百个一模一样的getName 方法，还不得不为getName 分配内存空间，即重复又浪费空间。那有没有办法实现：**同一个功能，让每一个实例对象都访问同一个方法呢？办法肯定是有的，那就是原型对象。**

在JavaScript中，每当定义一个函数数据类型(普通函数、类)时候，都会天生自带一个prototype属性，这个属性指向一个对象，即函数的原型对象，简称对象。

为了实现同一个功能，让每一个实例对象都访问同一个方法，结合原型，我们可以按照以下步骤实现：

1. 声明一个构造函数，此时javascript内部会帮我们生成构造函数的原型对象
2. 使用prototype属性去访问构造函数的原型对象，并选择性的将一些属性和方法挂载到原型对象上
3. 通过new构造函数出来的实例都会有一个\_\_proto\_\_属性，该属性指向构造函数的原型对象，通过这个属性，让实例对象也能够访问原型对象上的方法。因此，当所有的实例都能够通过\_\_proto\_\_属性访问到原型对象时，原型对象的方法与属性就变成了共有方法与属性。

看例子：

function Person(name, age) {

this.name = name;

this.age = age;

}

// 通过prototye属性，将方法挂载到原型对象上

Person.prototype.getName = function() {

return this.name;

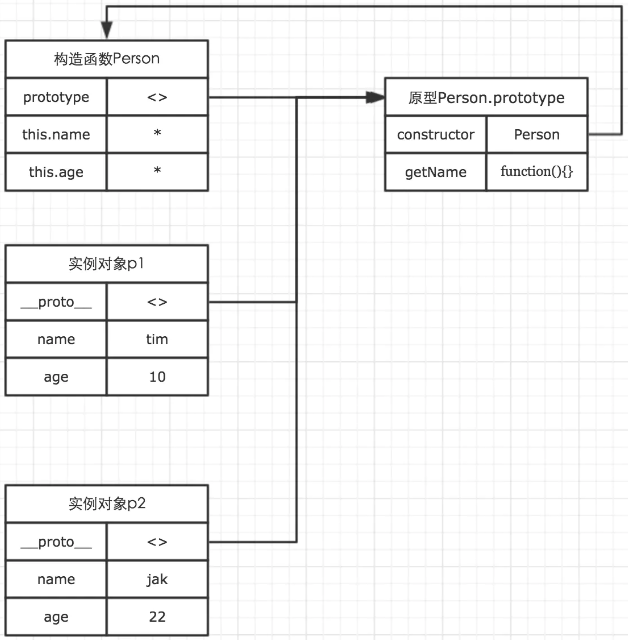
}

var p1 = new Person('tim', 10);

var p2 = new Person('jak', 22);

console.log(p1.getName === p2.getName); // true

用图展示：



通过该图，我们可以做出以下总结：

1. 构造函数的prototype属性与所有实例对象的\_\_proto\_\_都指向原型对象
2. 原型对象的constructor指向构造函数
3. 实例对象实际上对前面我们所说的中间对象的复制
4. 构造函数中的属性和方法通过this，被某个生成的实例对象所独有
5. 通过原型声明的属性与方法，可以被所有实例对象访问，是共有属性与方法

## 原型链

**属性与方法的访问顺序**

通过构造函数生成的实例，如果访问该实例的属性或者方法的时候，会访问实例对象的属性和方法，如果实例对象没有的话就会去原型对象上查找。

function Person(name, age) {

this.name = name;

this.age = age;

this.getName = function() {

console.log('this is constructor.');

}

}

Person.prototype.getName = function() {

return this.name;

}

Person.prototype.getAge = function() {

console.log(this.age)

return this.age;

}

var p1 = new Person('tim', 10);

p1.getName(); // this is constructor.

p1.getAge(); // 10

在上述代码中，getName 方法实例对象与原型对象都有，则优先访问实例对象的。访问getAge的时候，由于实例对象没有，则访问原型对象Person.prototype的。

为了判断方法的存在，javascript提供了一些方法去做以下判断：

1. hasOwnProperty()来检查对象自身中是否含有该属性
2. 使用in检查对象中是否含有某个属性时，如果对象中没有但是原型中有，也会返回true

看示例：

function Person() {}

Person.prototype.a = 123;

Person.prototype.sayHello = function () {

alert("hello");

};

var person = new Person()

console.log(person.a)//123

console.log(person.hasOwnProperty('a'));//false

console.log('a'in person)//true

person实例中没有a这个属性，从 person 对象中找不到 a 属性就会从 person 的原型也就是 person.\_\_proto\_\_ ，也就是 Person.prototype中查找，很幸运地得到a的值为123。那假如 person.\_\_proto\_\_中也没有该属性，又该如何查找？

当读取实例的属性时，如果找不到，就会查找与对象关联的原型中的属性，如果还查不到，就去找原型的原型，一直找到最顶层Object为止。Object是JS中所有对象数据类型的基类(最顶层的类)在Object.prototype上没有\_\_proto\_\_这个属性。

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

在JavaScript中万物都是对象，对象和对象之间也有关系，并不是孤立存在的。对象之间的继承关系，在JavaScript中是通过prototype对象指向父类对象，直到指向Object对象为止，这样就形成了一个原型指向的链条，专业术语称之为原型链。

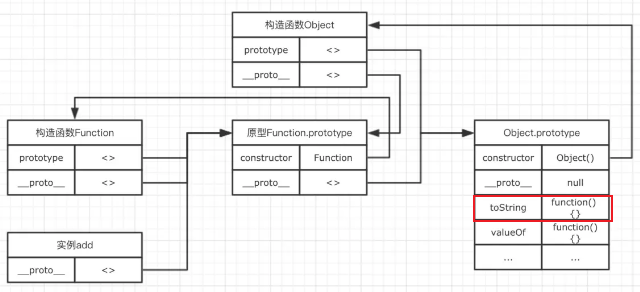
正如我们例子展示，当我们访问实例对象的一个属性或方法时，它会先在实例对象自身中寻找，如果有则直接使用，如果没有则会去原型对象中寻找，如果找到则直接使用。如果没有则去原型的原型中寻找,直到找到Object对象的原型，Object对象的原型没有原型，如果在Object原型中依然没有找到，则返回undefined。

比较具体的表现是，我们定义函数之后，函数都有一个toString方法，那么这个方法是怎么来的呢？

function add () {}

console.log(add.toString()) // function add () {}

用图展示下：



add是Function创建的实例对象，而Function的原型对象同时又是Object的实例，这个就构成一条原型链。add实例对象能够通过原型链，单向的查找访问到处于原型链上对象的所有属性与方法。这也是add最终能够访问到处于Object原型对象上的toString方法的原因。

## 继承

当我们新建一个构造函数A，只是对之前的构造函数B做了拓展，想要既有B函数的所有属性和方法，又能在A函数上新增属性和方法，此时我们可以通过继承的方式实现。

通过上面的讲解，我们知道了可以结合构造函数和原型对象来创建一个实例对象，因此想要实现继承功能，我们需要根据构造函数与原型对象的不同而采取不同的策略。

首先，我们声明一个Person对象作为父级，我们要实现子级Child继承Person所有的属性和方法。

function Person(name, age) {

this.name = name;

this.age = age;

}

Person.prototype.getName = function() {

return this.name;

}

**构造函数的继承：**

function Child(name, age, job) {

Person.call(this, name, age);

this.job = job;

}

call改变了Person函数中的this指向，使得Child也有了Person函数中定义的属性与方法。

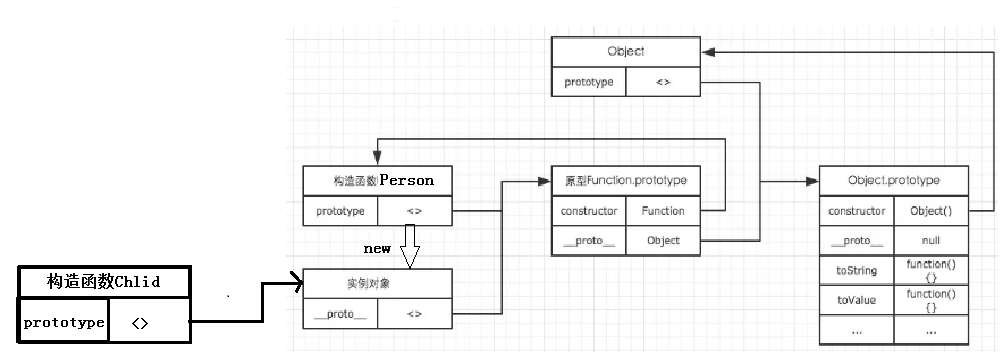
**原型的继承**

// 继承原型

Child.prototype = new Person(name, age);

// 添加更多方法

Child.prototype.getLive = function() {}



更简单的方式，我们可以封装一个方法来实现原型的继承：

function create(proto, options) {

// 创建一个空对象

var tmp = {};

// 让这个新的空对象成为父类对象的实例

tmp.\_\_proto\_\_ = proto;

// 传入的方法都挂载到新对象上，新的对象将作为子类对象的原型

Object.defineProperties(tmp, options);

return tmp;

}

使用方式：

Chlid.prototype = create(Person.prototype, {

// 不要忘了重新指定构造函数

constructor: {

value: Chlid

}

getGrade: {

value: function() {

return this.grade

}

}

})

验证一下我们这里实现的继承是否正确

var s1 = new Chlid('ming', 22, 5);

console.log(s1.getName()); // ming

console.log(s1.getAge()); // 22

console.log(s1.getGrade()); // 5

s1实例访问对应的方法都没问题，验证通过。

实际上，在ECMAScript5中直接提供了一个Object.create方法来完成我们上面自己封装的create的功能。因此我们可以直接使用Object.create.

Student.prototype = create(Person.prototype, {

// 不要忘了重新指定构造函数

constructor: {

value: Student

}

getGrade: {

value: function() {

return this.grade

}

}

})

## 小结

至此，原型、原型链、构造函数的内容已经分析完毕，小结一下：

1. 每个对象拥有一个原型对象，通过 \_\_proto\_\_ 指针指向上一个原型 ，并从中继承方法和属性，同时原型对象也可能拥有原型，这样一层一层，最终指向 null，这种关系被称为原型链

2.当访问一个对象的属性或方法时，它不仅仅在该对象上查找，还会查找该对象的原型，以及该对象的原型的原型，一层一层向上查找，直到找到一个名字匹配的属性或方法或到达原型链的末尾（null）。

3.原型链的构建依赖于 \_\_proto\_\_，一层一层最终链接到 null。

4.原型链继承的本质是重写原型对象，代之以一个新类型的实例