原型、原型链、继承

原型是什么?

prototype 属性

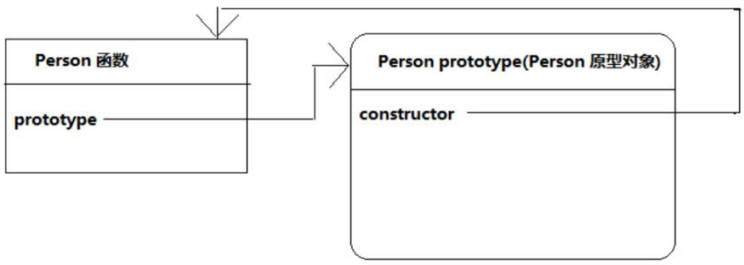
原型的作用就是用来存放公用的方法或者属性的，这么理解最简单，JavaScript 的继承就是基于原型的继承。

- prototype 本质上还是一个 JavaScript 对象；
- 每个函数都有一个默认的 prototype 属性；
- 通过 prototype 我们可以扩展 Javascript 的内建对象
- 通过原型我们可以达到继承的目的

那 prototype 属性的作用又是什么呢？它的作用就是包含可以由特定类型的所有实例共享的属性和方法，也就是让该函数所实例化的对象们都可以找到公用的属性和方法。任何函数在创建的时候，其实会默认同时创建该函数的 prototype 对象。

1.1 函数的原型对象

/*声明一个函数，则这个函数默认会有一个属性叫 prototype 。而且浏览器会自动按照一定的规则创建一个对象，这个对象就是这个函数的原型对象，prototype 属性指向这个原型对象。这个原型对象有一个属性叫 constructor 指向了这个函数
注意：原型对象默认只有属性：constructor。其他都是从 Object 继承而来，暂且不用考虑。*/
function Person () { }
下面的图描述了声明一个函数之后发生的事情



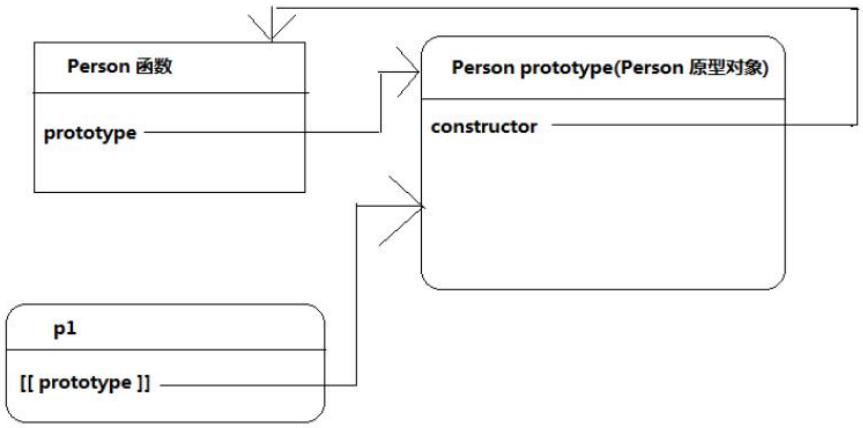
1.2 使用构造函数创建对象

当把一个函数作为构造函数 (理论上任何函数都可以作为构造函数) 使用 new 创建对象的时候，那么这个对象就会存在一个默认的不可见的属性，来指向了构造函数的原型对象。 **这个不可见的属性我们一般用 [[prototype]] 来表示，只是这个属性没有办法直接访问到。**

实际上，该属性在 ES 标准定义中的名字应该是`[[Prototype]]`，具体实现是由浏览器代理自己实现，谷歌浏览器的实现就是将`[[Prototype]]`命名为`__proto__`，大家清楚这个标准定义与具体实现的区别即可

```
function Person () { }
/*利用构造函数创建一个对象，则这个对象会自动添加一个不可见的属性 [[prototype]], 而且这个属性指向了构造函数的原型对象。*/
var p1 = new Person();
```

以上代码表示创建一个构造函数 Person()，并用 new 关键字实例化该构造函数得到一个实例化对象 p1。这里稍微补充一下 new 操作符将函数作为构造器进行调用时的过程：函数被调用，然后新创建一个对象，并且成了函数的上下文，最后返回该新对象的引用。



- 从上面的图示中可以看到，创建 p1 对象虽然使用的是 Person 构造函数，但是对象创建出来之后，这个 p1 对象其实已经与 Person 构造函数没有任何关系了，p1 对象的[[prototype]]属性指向的是 Person 构造函数的原型对象。
- 如果使用 new Person()创建多个对象，则多个对象都会同时指向 Person 构造函数的原型对象。
- 我们可以手动给这个原型对象添加属性和方法，那么 p1,p2,p3…这些对象就会共享这些在原型中添加的属性和方法。
- 如果我们访问 p1 中的一个属性 name，如果在 p1 对象中找到，则直接返回。如果 p1 对象中没有找到，则直接去 p1 对象的[[prototype]]属性指向的原型对象中查找，如果查找到则返回。(如果原型中也没有找到，则继续向上找原型的原型—原型链)。
- 如果通过 p1 对象添加了一个属性 name，则 p1 对象来说就屏蔽了原型中的属性 name。 换句话说：在 p1 中就没有办法访问到原型的属性 name 了。
- 通过 p1 对象只能读取原型中的属性 name 的值，而不能修改原型中的属性 name 的值。
  p1.name = “李四”；并不是修改了原型中的值，而是在 p1 对象中给添加了一个属性 name。

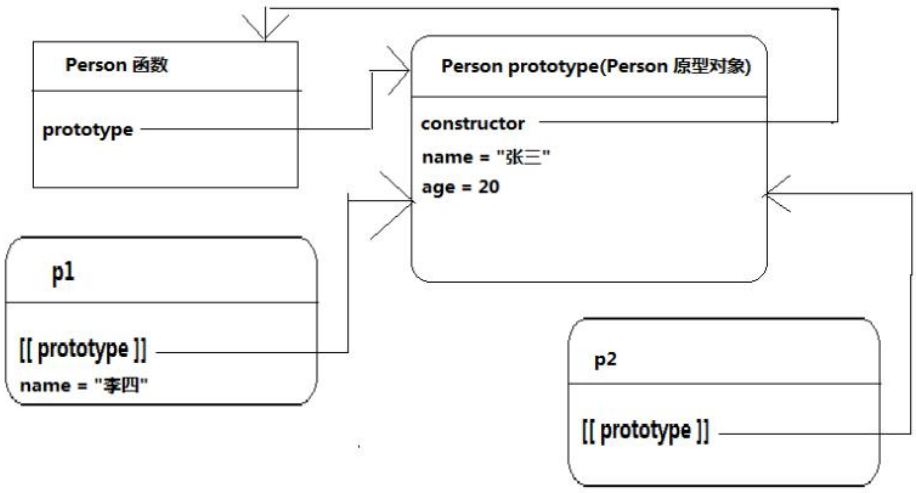
看下面的代码：

```
<body>
  <script type="text/javascript">
```

```
        function Person () { }
        // 可以使用 Person.prototype 直接访问到原型对象
        //给 Person 函数的原型对象中添加一个属性 name 并且值是 "张三"
        Person.prototype.name = "张三";
        Person.prototype.age = 20;

        var p1 = new Person();
/*访问 p1 对象的属性 name，虽然在 p1 对象中我们并没有明确的添加属性 name，但是 p1 的 [[prototype]] 属性指向的原型中有 name 属性，所以这个地方可以访问到属性 name 就值。
注意：这个时候不能通过 p1 对象删除 name 属性，因为只能删除在 p1 中删除的对象。*/
            console.log(p1.name); // 张三
            var p2 = new Person();
            console.log(p2.name); // 张三 都是从原型中找到的，所以一样。
            console.log(p1.name === p2.name); // true

            // 由于不能修改原型中的值，则这种方法就直接在 p1 中添加了一个新的属性 name，然后在 p1 中无法再访问到
            //原型中的属性。
            p1.name = "李四";
            console.log("p1：" + p1.name);
            // 由于 p2 中没有 name 属性，则对 p2 来说仍然是访问的原型中的属性。
            console.log("p2:" + p2.name); // 张三
    </script>
</body>
```



与原型有关的几个属性和方法

2.1 prototype 属性

prototype 在于构造函数中 (任意函数中都有，只是不是构造函数的时候 prototype 我们不关注而已)，他指向了这个构造函数的原型对象

2.2 constructor 属性

constructor 属性存在于原型对象中，他指向了构造函数

constructor 属性也是对象才拥有的，它是从一个对象指向一个函数，含义就是指向该对象的构造函数，每个对象都有构造函数（本身拥有或继承而来，继承而来的要结合__proto__属性查看会更清楚点）

什么是 instanceof

我们经常使用 instanceof 就是判断一个实例是否属于某种类型，instanceof 可以在继承关系中用来判断一个实例是否属于它的父类型

简单实例：

```
function Foo(){}
function Bar(){}
Bar.prototype = new Foo()
let obj = new Bar()
obj instanceof  Bar //true  //验证孩子是不是亲生的
obj instanceof  Foo //true
```

```
function Foo(){};
Foo instanceof Function //true
Foo instanceof Foo //false
```

```
function Person () { }
console.log(Person.prototype.constructor === Person);       // true 找父母
var p1 = new Person();
//使用 instanceof 操作符可以判断一个对象的类型。
p1 instanceof Person ; // true   //验证孩子是不是亲生的
Person instanceof  Function //true
p1 instanceof Person //false
```

我们根据需要，可以 Person.prototype 属性指定新的对象，来作为 Person 的原型对象。
但是这个时候有个问题，新的对象的 constructor 属性则不再指向 Person 构造函数了。

```
<script type="text/javascript">
        function Person () { }
        Person.prototype .say = function(){ console.log( “hello” );}

        var p1 = new Person();
        console.log(p1.say);  // hello

        console.log(p1 instanceof Person); // true
        console.log(Person.prototype.constructor === Person); //true


       //直接给 Person 的原型指定对象字面量。则这个对象的 constructor 属性不再指向 Person 函数
        Person.prototype = {
                name:"song",
                age:20
        };
console.log(Person.prototype.constructor === Person); //false

//如果 constructor 对你很重要，你应该在 Person.prototype 中添加一行这样的代码：
/*
  Person.prototype = {
      constructor : Person    //让 constructor 重新指向 Person 函数
}*/
</script>
```

2.3 __proto__ 属性(注意：左右各是 2 个下划线)

用构造方法创建一个新的对象之后，这个对象中默认会有一个不可访问的属性 [[prototype]] , 这个属性就指向了构造方法的原型对象。

但是在某些浏览器中，也提供了对这个属性[[prototype]]的访问(chrome 浏览器和火狐浏览器)。访问方式：
p1.__proto__

但是开发者尽量不要用这种方式去访问，因为操作不慎会改变这个对象的继承原型链

①__proto__和 constructor 属性是对象所独有的；
② prototype 属性是函数所独有的。但是由于 JS 中函数也是一种对象，所以函数也拥有__proto__和 constructor 属性，这点是致使我们产生困惑的很大原因之一。

var p1 = new Object({name: “laney” });

Person.prototype.__proto__ ==Object.prototype //true
Person.prototype.__proto__ .__proto__ ==null //true

第一，这里我们仅留下 `__proto__` 属性，它是**对象所独有的**，可以看到`__proto__`属性都是由**一个对象指向一个对象**，即指向它们的原型对象（也可以理解为父对象）

那么这个属性的作用是什么呢？

它的作用就是当访问一个对象的属性时，如果该对象内部不存在这个属性，那么就会去它的__proto__属性所指向的那个对象（可以理解为父对象）里找，如果父对象也不存在这个属性，则继续往父对象的__proto__属性所指向的那个对象（可以理解为爷爷对象）里找，如果还没找到，则继续往上找…直到原型链顶端 null（可以理解为原始人。。。），再往上找就相当于在 null 上取值，会报错（可以理解为，再往上就已经不是“人”的范畴了，找不到了，到此结束，null 为原型链的终点），由以上这种通过__proto__属性来连接对象直到 null 的一条链即为我们所谓的**原型链**。

2.4 hasOwnProperty() 方法

我们用去访问一个对象的属性的时候，这个属性既有可能来自对象本身，也有可能来自这个对象的[[prototype]]属性指向的原型。
那么如何判断这个对象的来源呢?

hasOwnProperty 方法，可以判断一个属性是否来自对象本身。

```
function Person () { }
Person.prototype.name = "志玲";
var p1 = new Person();
p1.sex = "女";
//sex 属性是直接在 p1 属性中添加，所以是 true
alert("sex 属性是对象本身的：" + p1.hasOwnProperty("sex"));
// name 属性是在原型中添加的，所以是 false
alert("name 属性是对象本身的：" + p1.hasOwnProperty("name"));
//  age 属性不存在，所以也是 false
alert("age 属性是存在于对象本身：" + p1.hasOwnProperty("age"));
```

原型模式适合封装方法，构造函数模式适合封装属性，综合两种模式的优点就有了组合模式。

```
//在构造方法内部封装属性
 function Person(name, age) {
    this.name = name;
```

```
    this.age = age;
}
//在原型对象内封装方法
Person.prototype.eat = function (food) {
        alert(this.name + "爱吃" + food);
}
Person.prototype.play = function (playName) {
        alert(this.name + "爱玩" + playName);
}
var p1 = new Person("李四", 20);
var p2 = new Person("张三", 30);
p1.eat("苹果");
p2.eat("香蕉");
p1.play("志玲");
p2.play("凤姐");
```

创建对象和生成原型链(不要)

使用构造函数创建对象

```
function Person(name,age){
    this.name = name;
     this.age = age;
  }
var p1 = new Person();
```

使用原型创建对象

```
function Person(name){
        this.name=‘laney’
    }
Person.prototype.sayHello =function(){
     console.log(this.name)
};
var p2 = new Person();
console.log(Person.prototype.isPrototypeOf(p2));
```

//知识点：isPrototypeOf ，instanceof
检测一个对象是否是另一个对象的原型。或者说一个对象是否被包含在另一个对象的原型链中

```
var p = {x:1};//定义一个原型对象
var o = Object.create(p);//使用这个原型创建一个对象
p.isPrototypeOf(o);//=>true：o 继承 p,检测 p 是否是 o 的原型
Object.prototype.isPrototypeOf(p);//=> true p 继承自 Object.prototype
```

使用 class 关键字创建的对象，es6 里的内容，可以不讲

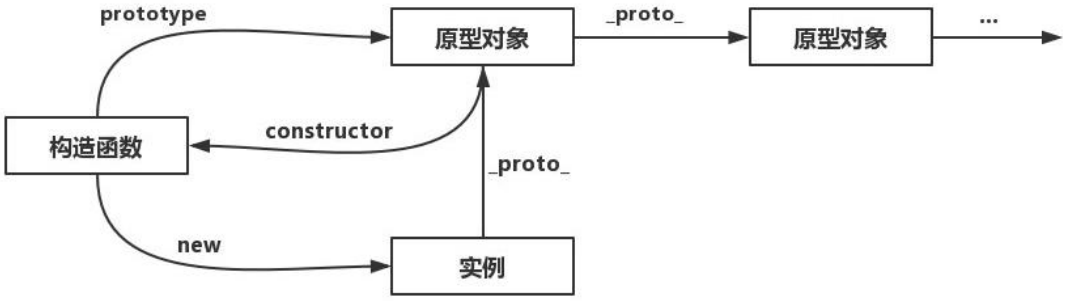
```
class Person {
    constructor(name){
      this.name = name
    }
     getName(){
          console.log(this.name)
    }
}
class Student extends Person {
        constructor(name,age){
                super(name);
                this.age = age
        }
        getAge(){
                console.log(this.age);
        }
    getName2(){
```

```
        console.log(this.name)
    }
}
var p1 = new Person('laney');
var p2 = new Person('song');
```

3. JS 继承的实现方式

函数对象都有 prototype 属性，原型链是实现继承的主要方法，基本思想就是利用原型让一个引用类型继承另一个引用类型的属性和方法。
要理解原型链就要明白原型对象、构造函数、实例，三者之间的关系。

每创建一个函数，该函数就会自动带有一个 prototype 属性。该属性是个指针，指向了一个对象，我们称之为 原型对象。
原型对象上默认有一个属性 constructor，指向其相关联的构造函数。
通过构造函数产生的实例，都有一个内部属性，指向了原型对象。所以实例能够访问原型对象上的所有属性和方法。



原型链说明

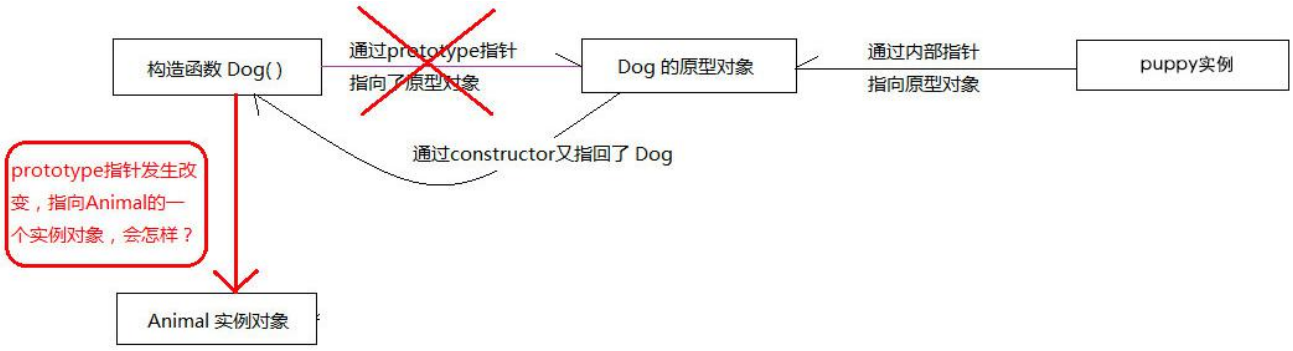
构造函数：
function A() {}
A.prototype.constructor ===A
A.prototype.__proto__ === Object.prototype

1. 原型链继承

```
function Dog (name) {
   this.name = name;
   this.type = 'Dog';
}
Dog.prototype.speak = function () {
       console.log(‘wang wang');
}
var puppy = new Dog('小黑');
puppy.speak();  //wang
```

以上代码定义了一个构造函数 Dog(), Dog.prototype 指向的原型对象，其自带的属性 construtor 又指回了 Dog，即 Dog.prototype.constructor == Dog. 实例 puppy 由于其内部指针指向了该原型对象，所以可以访问到 speak 方法。
Dog 的原型示意图

Dog.prototype 只是一个指针，指向的是原型对象，但是这个原型对象并不特别，它也只是一个普通对象。假设说，这时候，我们让 Dog.protptype 不再指向最初的原型对象，而是另一个类 （Animal）的实例，情况会怎样呢？




```
// 定义一个动物类
    function Animal (category) {
        // 属性
        this.category = category || 'Animal';
        This.testArr=[‘老虎’,’狮子’];
        // 实例方法
        this.sleep = function(){
            console.log(this.category + '正在睡觉！ ');
        }
    }
    // 原型方法
    Animal.prototype.eat = function(food) {
        console.log(this.category + '正在吃：' + food);
    };
    // 1. 原型继承
    function Dog (name) {
        this.name = name;
        this.type = 'Dog';
    }
    //改变 Dog 的 prototype 指针，指向一个 Animal 实例
    Dog.prototype = new Animal();  //Dog.prototype = Object.create(Animal.prototype);
    Dog.prototype.constructor = Dog;

    Dog.prototype.speak = function () {
            console.log('wang wang ');
    }
```

```
var puppy = new Dog('xiao hei');
    puppy.speak();  //wang wang
    console.log(puppy.category);
    console.log(puppy.testArr);
    console.log('------------');
    puppy.testArr.push('狮子');
    puppy.category = '虫类';
```

解释一下以上代码：

1. 定义一个父类的构造函数 Animal, 包括实例属性 category 和实例方法 sleep, 以及原型方法 eat ,
2. 再定义子类的构造函数 Dog , 包括实例属性 name 和 type
3. 情况有变， 把 Dog 原型对象覆盖成 animal 实例
4. 再添加 Dog 的原型方法
5. 实例化 Dog 对象，实现继承

通过原型链的方式，实现 Dog 继承 Animal 的所有属性和方法

```
var puppy2 = new Dog('xiao bai');
    console.log(puppy2.category);
    console.log(puppy2.testArr);
    console.log('------------');

    puppy2.category = '鸟';
    puppy2.testArr.push('老虎');

    var puppy3 = new Dog('xiao cong');
    console.log(puppy3.category);
    console.log(puppy3.testArr);
    console.log( ‘------------');
```

原型链继承缺点：

1. 要想为子类新增属性和方法，必须要在 new Animal()这样的语句之后执行，不能放到构造器中
2. 无法实现多继承

3. 创建子类实例时，无法向父类构造函数传参

4. 原型上任何类型的属性值都不会通过实例被重写，但是引用类型的属性值会受到实例的影响而修改

2. 构造继承

```
function Animal(cagegory){
        this.cagegory = cagegory;
        this.sleep = function(){
        console.log(this.cagegory+’正在睡觉’);
        }
}
 function DogA (cagegory,name) {
      Animal.call(this,cagegory);
      this.name = name;
      this.type = 'Dog';
   }
//DogA.prototype = new Animal();

DogA.prototype.constructor = DogA;
   var dog1 = new DogA(‘pang');
```

1 . 只能继承父类的实例属性和方法，不能继承原型属性/方法
2. 无法实现函数复用，每个子类都有父类实例函数的副本，影响性能

3. 组合继承

核心：为父类实例添加新特性，作为子类实例返回

```
function Animal(cagegory){
        this.cagegory = cagegory;
        this.sleep = function(){
        console.log(this.cagegory+’正在睡觉’);
```

```
        }
}
Animal.prototype.run= function(){ console.log(‘run’);}
 function DogB(category,name){
       Animal.call(this,category);
       this.name = name;
       this.type = 'Dog';
   }
   DogB.prototype = new Animal();
   DogB.prototype.constructor = DogB;

   var dog2 = new DogB('动物','xiao huang');
   console.log(dog2);
```

缺点：调用了两次父类构造函数，生成了两份实例(仅仅多消耗了一点内存)

4. create 组合继承

```
function Animal(cagegory){
        this.cagegory = cagegory;
        this.sleep = function(){
        console.log(this.cagegory+’正在睡觉’);
        }
}
Animal.prototype.run= function(){ console.log(‘run’);}

 function DogB(category,name){
       Animal.call(this,category);
       this.name = name;
       this.type = 'Dog';
   }
DogB.prototype = Object.create(Animal.prototype);
DogB.prototype.constructor = DogB;

var dog2 = new DogB('动物','xiao huang');
console.log(dog2);
console.log(dog2 instanceof DogB)
console.log(dog2 instanceof Animal)
```

如何实现多继承呢 ？

```
function Animal1(){
        this.sleep = function(){
        console.log(this.cagegory+’正在睡觉’);
        }
}
function Animal2(cagegory){
        this.cagegory = cagegory;
}
Animal2.prototype.run= function(){ console.log(‘run’);}

 function DogB(category,name){
       Animal1.call(this);
      Animal2.call(this,category);

       this.name = name;
       this.type = 'Dog';
   }

DogB.prototype = Object.create(Animal1.prototype);

Object.assign(DogB.prototype, Animal2.prototype);
DogB.prototype.constructor = DogB;
```