在JavaScript中，对象的创建可以脱离类型（class free），通过字面量的方式可以很方便的创建出自定义对象。

另外，JavaScript中拥有原型这个强大的概念，当对象进行属性查找的时候，如果对象本身内找不到对应的属性，就会去搜索原型链。所以，结合原型和原型链的这个特性，JavaScript就可以用来实现对象之间的继承了。

下面就介绍一下JavaScript中的一些常用的继承方式。

# 原型链继承

由于原型链搜索的这个特性，在JavaScript中可以很方便的通过原型链来实现对象之间的继承。

下面看一个例子：

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

Person.prototype.getInfo = function(){
    console.log(this.name + " is " + this.age + " years old!");
}


function Teacher(staffId){
    this.staffId = staffId;
}

Teacher.prototype = new Person();

var will = new Teacher(1000);
will.name = "Will";
will.age = 28;
will.getInfo();
// Will is 28 years old!

console.log(will instanceof Object);
// true
console.log(will instanceof Person);
// true
console.log(will instanceof Teacher);
// true

console.log(Object.prototype.isPrototypeOf(will));
// true
console.log(Person.prototype.isPrototypeOf(will));
// true
console.log(Teacher.prototype.isPrototypeOf(will));
// true
```

在这个例子中，有两个构造函数"Person"和"Teacher"，通过"Teacher.prototype = new Person()"语句创建了一个"Person"对象，并且设置为"Teacher"的原型。

通过这种方式，就实现了"Teacher"继承"Person"，"will"这个对象可以成功的调用"getInfo"这个属于"Person"的方法。

在这个例子中，还演示了通过"instanceof"操作符和"isPrototypeOf()"方法来查看对象和原型之间的关系。

对于原型链继承，下面看看其中的一些细节问题。

## **constructor**属性

对于所有的JavaScript原型对象，都有一个"constructor"属性，该属性对应用来创建对象的构造函数。

对于"constructor"这个属性，最大的作用就是可以帮我们标明一个对象的"类型"。

在JavaScript中，当通过"typeof"查看Array对象的时候，返回的结果是"object"。这个我们的预期结果，所以如果要判对一个对象到底是不是Array类型，就可以结合"constructor"属性得到想要的结果。

```
function isArray(myArray) {
    return myArray.constructor.toString().indexOf("Array") > -1;
}

var arr = []
console.log(typeof arr);
// object
console.log(isArray(arr));
// true
```

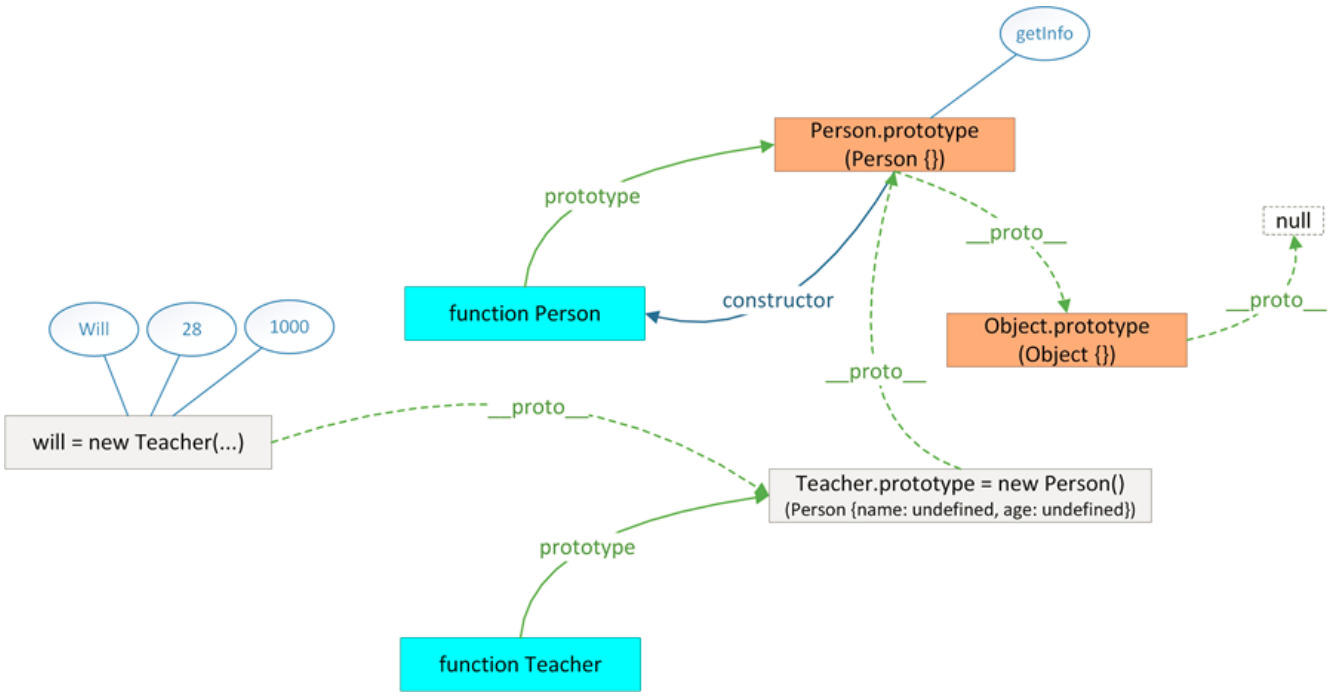
现在回到前面的例子，查看一下对象"will"的原型和构造函数：

```
> will.__proto__
◂ ▼Person {name: undefined, age: undefined}
    age: undefined
    name: undefined
  ▶ __proto__: Person
> will.__proto__.constructor
◂ function Person(name, age){
      this.name = name;
      this.age = age;
  }
> Teacher.prototype
◂ ▼Person {name: undefined, age: undefined}
    age: undefined
    name: undefined
  ▶ __proto__: Person
> Person.prototype
◂ ▼Person {}
  ▶ constructor: function Person(name, age)
  ▶ getInfo: function ()
  ▶ __proto__: Object
```

从这个结果可以看到，"will"的原型是"Person {name: undefined, age: undefined}"（通过new Person()构造出来的对象），"will"的构造函数是"function Person"。

等等，**"will"**不是通过**"Teacher"**创建出来的对象么？为什么构造函数对于的是**"function Person"**，而不是**"function Teacher"**？

下面，根据前面的例子绘制一张对象关系图，从而分析一下继承关系以及"constructor"属性：




图中给出了各种对象之间的关系，有几点需要注意的是：

- "Teacher.prototype"这个原型对象是通过"Person"构造函数创建出来的一个对象"Person {name: undefined, age: undefined}"
- 对象"will"创建了自己的"name"和"age"属性，并没有使用父类对象的，而是覆盖了父类的**"name"**和**"age"**属性
- 通过"will"访问"constructor"这个属性的时候，先找到了"Teacher.prototype"这个对象，然后找到"Person.prototype"，通过原型链查找访问到了**"constructor"**属性对应的**"function Person"**

### 重设**constructor**

为了解决上面的问题，让子类对象的"constructor"属性对应正确的构造函数，我们可以重设子类原型对象的"constructor"属性。

一般来说，可以简单的使用下面代码来重设"constructor"属性：

```
Teacher.prototype.constructor = Teacher;
```

但是通过这种方式重设"constructor"属性会导致它的[[Enumerable]]特性被设置为 true。默认情况下，原生的"constructor"属性是不可枚举的。

因此如果使用兼容 ECMAScript 5 的 JavaScript 引擎，就可以使用"Object.defineProperty()"：

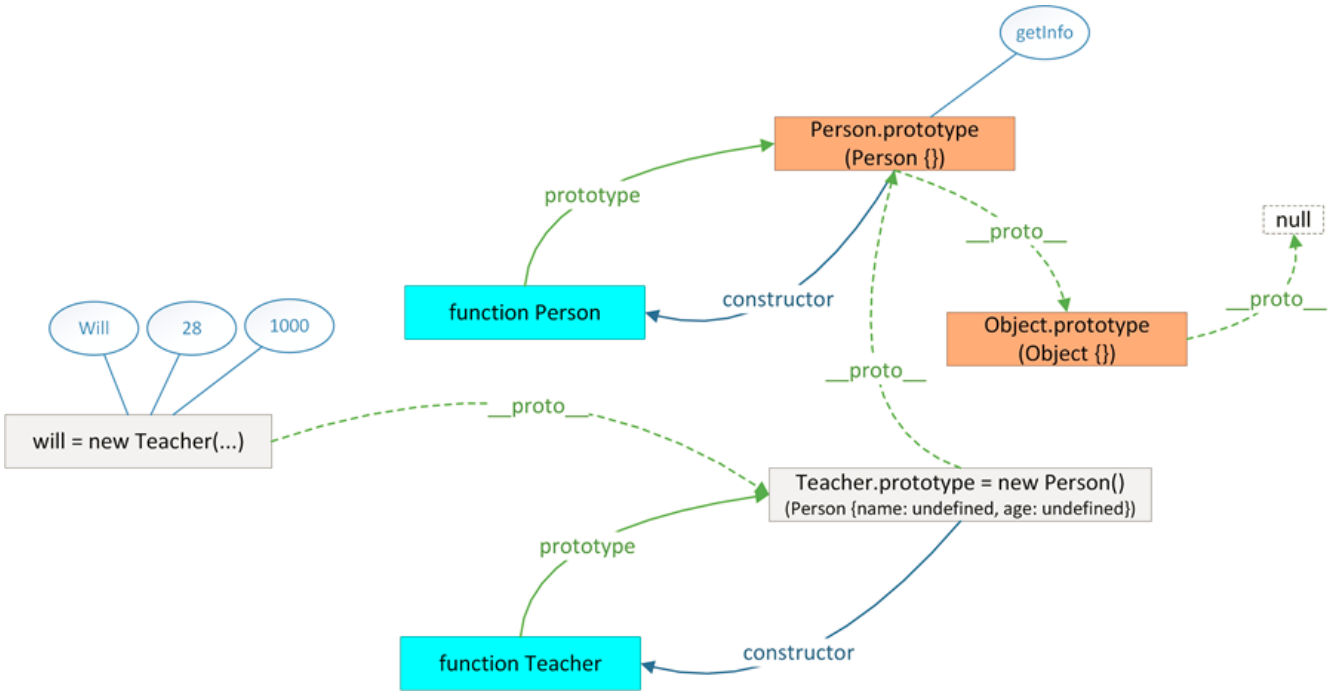
```
Object.defineProperty(Teacher.prototype, "constructor", {
    enumerable: false,
    value: Teacher
});
```

通过下面的结果可以看到：

```
> Object.defineProperty(Teacher.prototype, "constructor", {
      enumerable: false,
      value: Teacher
  });
< ▶ Person {name: undefined, age: undefined}
> will.__proto__
< ▼ Person {name: undefined, age: undefined}
    age: undefined
  ▶ constructor: function Teacher(staffId)
    name: undefined
  ▶ __proto__: Person
> will.__proto__.constructor
< function Teacher(staffId){
      this.staffId = staffId;
  }
```

通过这个设置，对象"will" 的"constructor"属性就指向了正确的"function Teacher"。

这时的对象关系图就变成了如下，跟前面的关系图比较，唯一的区别就是"Teacher.prototype"对象多了一个"constructor"属性，并且这个属性指向"function Teacher"：



## 原型的动态性

原型对象是可以修改的，所以，当创建了继承关系之后，我们可以通过更新子类的原型对象给子类添加特有的方法。

例如通过下面的方式就给子类添加了一个特有的"getId"方法。

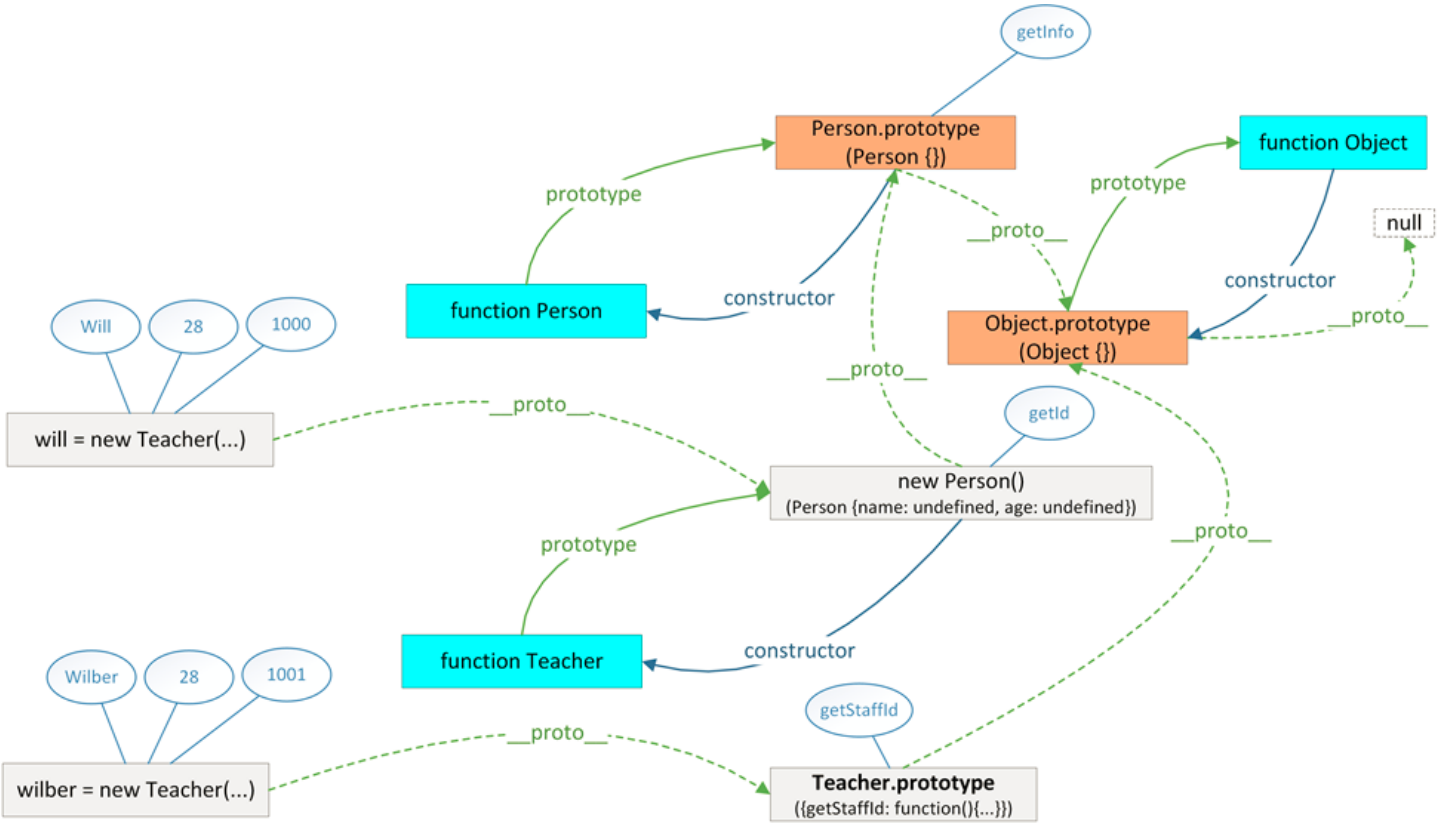
```
Teacher.prototype.getId = function(){
    console.log(this.name + "'s staff Id is " + this.staffId);
}

will.getId();
// Will's staff Id is 1000
```

但是，一定要区分原型的修改和原型的重写。如果对原型进行了重写，就会产生完全不同的效果。

下面看看如果对"Teacher"的原型重写会产生什么效果，为了分清跟前面代码的顺序，这里贴出了完整的代码：

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

Person.prototype.getInfo = function(){
    console.log(this.name + " is " + this.age + " years old!");
}


function Teacher(staffId){
    this.staffId = staffId;
}

Teacher.prototype = new Person();
Object.defineProperty(Teacher.prototype, "constructor", {
    enumerable: false,
    value: Teacher
});

var will = new Teacher(1000);
will.name = "Will";
will.age = 28;

// 更新原型
Teacher.prototype.getId = function(){
    console.log(this.name + "'s staff Id is " + this.staffId);
}

will.getId();
// Will's staff Id is 1000

// 重写原型
Teacher.prototype = {
    getStaffId: function(){
        console.log(this.name + "'s staff Id is " + this.staffId);
    }
}

will.getInfo();
// Will is 28 years old!
will.getId();
// Will's staff Id is 1000
console.log(will.__proto__);
// Person {name: undefined, age: undefined}
console.log(will.__proto__.constructor);
// function Teacher

var wilber = new Teacher(1001);
wilber.name = "Wilber";
wilber.age = 28;
// wilber.getInfo();
// Uncaught TypeError: wilber.getInfo is not a function(…)
wilber.getStaffId();
// Wilber's staff Id is 1001
console.log(wilber.__proto__);
// Object {}
console.log(wilber.__proto__.constructor);
// function Object() { [native code] }
```

经过重写原型之后情况更加复杂了，下面就来看看重写原型之后的对象关系图：




从关系图可以看到：

- 原型对象可以被更新，通过"Teacher.prototype.getId"给"will"对象的原型添加了"getId"方法
- 重写原型之后，在重写原型之前创建的对象的**"[[prototype]]"**属性依然指向原来的原型对象；在重写原型之后创建的对象的**"[[prototype]]"**属性将指向新的原型对象
- 对于重写原型前后创建的两种对象，对象的属性查找将搜索不同的原型链

## 组合继承

在通过原型链方式实现的继承中，父类和子类的构造函数相对独立，如果子类构造函数可以调用父类的构造函数，并且进行相关的初始化，那就比较好了。

这时就想到了JavaScript中的call方法，通过这个方法可以动态的设置this的指向，这样就可以在子类的构造函数中调用父类的构造函数了。

这样就有了组合继承这种方式：

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

Person.prototype.getInfo = function(){
    console.log(this.name + " is " + this.age + " years old!");
}

function Teacher(name, age, staffId){
    Person.call(this, name, age);        // 通过call方法来调用父类的构造函数进行初始化
    this.staffId = staffId;
}

Teacher.prototype = new Person();
Object.defineProperty(Teacher.prototype, "constructor", {
    enumerable: false,
    value: Teacher
});

var will = new Teacher("Will", 28, 1000);
will.getInfo();

console.log(will.__proto__);
// Person {name: undefined, age: undefined}
console.log(will.__proto__.constructor);
// function Teacher
```

在这个例子中，在子类构造函数"Teacher"中，直接通过"Person.call(this, name, age);"的方式调用了父类的构造函数，进而设置了"name"和"age"属性（但这里依旧是覆盖了父类的"name"和"age"属性）。

组合式继承是比较常用的一种继承方法，其背后的思路是使用原型链实现对原型属性和方法的继承，而通过借用构造函数来实现对实例属性的继承。这样，既通过在原型上定义方法实现了函数复用，又保证每个实例都有它自己的属性。

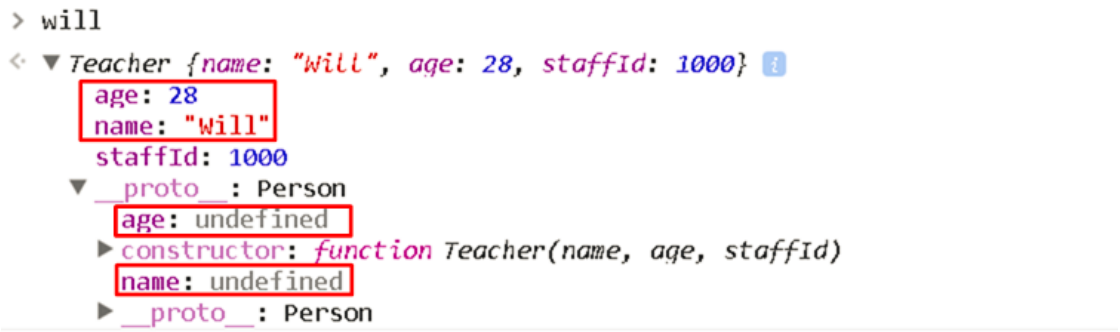
### 组合式继承的小问题

虽然组合继承是 JavaScript 比较常用的继承模式，不过通过前面组合继承的代码可以看到，它也有一些小问题。

首先，子类会调用两次父类的构造函数：

- 一次是在创建子类型原型的时候
- 另一次是在子类型构造函数内部

子类型最终会包含超类型对象的全部实例属性，但我们不得不在调用子类型构造函数时重写这些属性，从下图可以看到"will"对象中有两份"name"和"age"属性。




后面，我们会看到如何通过"寄生组合式继承"来解决组合继承的这个问题。

## 原型式继承

在前面两种方式中，都需要用到对象以及创建对象的构造函数（类型）来实现继承。

但是在JavaScript中，创建对象完全不需要定义一个构造函数（类型），通过字面量的方式就可以创建一个自定义的对象。

为了实现对象之间的直接继承，就有了原型式继承。

这种继承方式方法并没有使用严格意义上的构造函数，而是直接借助原型基于已有的对象创建新对象，同时还不必创建自定义类型（构造函数）。为了达到这个目的，我们可以借助下面这个函数：

```
function object(o){
    function F(){}
    F.prototype = o;
    return new F();
}
```

在 "object()"函数内部，先创建了一个临时性的构造函数，然后将传入的对象作为这个构造函数的原型，最后返回了这个临时类型的一个新实例。

下面看看使用"object()"函数实现的对象之间的继承：

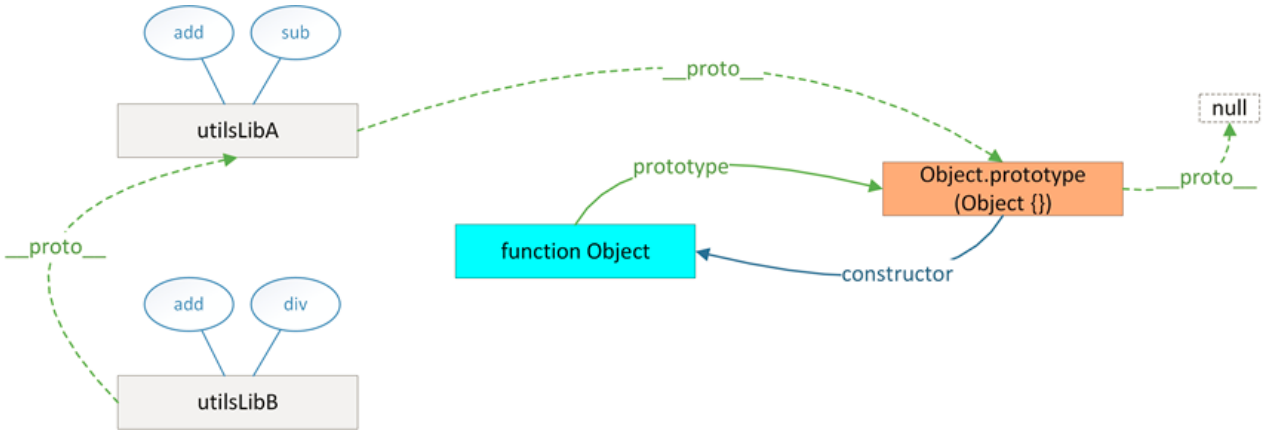
```
var utilsLibA = {
    add: function(){
        console.log("add method from utilsLibA");
    },
    sub: function(){
        console.log("sub method from utilsLibA");
    }
}

var utilsLibB = object(utilsLibA);

utilsLibB.add = function(){
    console.log("add method from utilsLibB");
}
utilsLibB.div = function(){
    console.log("div method from utilsLibB");
}

utilsLibB.add();
// add method from utilsLibB
```

```
utilsLibB.sub();
// sub method from utilsLibA
utilsLibB.div();
// div method from utilsLibB
```

通过原型式继承，基于"utilsLibA"创建了一个"utilsLibB"对象，并且可以正常工作，下面看看对象之间的关系：

```
> utilsLibA.__proto__
◂ ▶ Object {}
> utilsLibA.__proto__.constructor
◂ function Object() { [native code] }
> utilsLibB.__proto__
◂ ▼ Object {}
    ▶ add: function ()
    ▶ sub: function ()
    ▶ __proto__: Object
> utilsLibB.__proto__.constructor
◂ function Object() { [native code] }
> utilsLibB.__proto__ === utilsLibA
◂ true
```

通过"object()"函数的帮助，将"utilsLibB"的原型赋值为"utilsLibA"，对于这个原型式继承的例子，对象关系图如下，"utilsLibB"的"add"方法覆盖了"utilsLibA"的"add"方法：




## Object.create()

ECMAScript 5 通过新增 "Object.create()"方法规范化了原型式继承。这个方法接收两个参数：

- 一个用作新对象原型的对象
- 一个为新对象定义额外属性的对象（可选的）

在传入一个参数的情况下，"Object.create()"与 上面的"object"函数行为相同。关于更多"Object.create()"的内容，请参考MDN。

继续上面的例子，这次使用"Object.create()"来创建对象"utilsLibC"：

```
utilsLibC = Object.create(utilsLibA, {
    sub: {
        value: function(){
            console.log("sub method from utilsLibC");
        }
    },
    mult: {
        value: function(){
            console.log("mult method from utilsLibC");
        }
    },
})

utilsLibC.add();
// add method from utilsLibA
utilsLibC.sub();
```

```
// sub method from utilsLibC
utilsLibC.mult();
// mult method from utilsLibC
console.log(utilsLibC.__proto__);
// Object {add: (), sub: (), __proto__: Object}
console.log(utilsLibC.__proto__.constructor);
// function Object() { [native code] }
```

## 寄生式继承

寄生式继承是与原型式继承紧密相关的一种思路，寄生式继承的思路与寄生构造函数和工厂模式类似，即创建一个仅用于封装继承过程的函数，该函数在内部以某种方式来增强对象，最后再像真地是它做了所有工作一样返回对象。

以下代码示范了寄生式继承模式，其实就是封装"object()"函数的调用，以及对新的对象进行自定义的一些操作：

```
function create(o){
    var f= object(o);          // 通过原型式继承创建一个新对象
    f.run = function () {      // 以某种方式来增强这个对象
        return this.arr;
    };
    return f;                  // 返回对象
}
```

## 寄生组合式继承

所谓寄生组合式继承，即通过借用构造函数来继承属性，通过原型链的混成形式来继承方法。

其背后的基本思路是：不必为了指定子类型的原型而调用超类型的构造函数，我们所需要的无非就是父类型原型的一个副本而已。本质上，就是使用寄生式继承来继承父类型的原型，然后再将结果指定给子类型的原型。

注意在寄生组合式继承中使用的“inheritPrototype()”函数。

```
function object(o) {
    function F() {}
    F.prototype = o;
    return new F();
}

function inheritPrototype(subType, superType) {
    var prototype = object(superType.prototype);    // 创建对象
    prototype.constructor = subType;                // 增强对象，设置constructor属性
    subType.prototype = prototype;                  // 指定对象
}

function Person(name, age){
    this.name = name;
    this.age = age;
}
Person.prototype.getInfo = function(){
    console.log(this.name + " is " + this.age + " years old!");
}


function Teacher(name, age, staffId){
    Person.call(this, name, age)
    this.staffId = staffId;
}

inheritPrototype(Teacher, Person);

Teacher.prototype.getId = function(){
    console.log(this.name + "'s staff Id is " + this.staffId);
}


var will = new Teacher("Will", 28, 1000);
will.getInfo();
// Will is 28 years old!
will.getId();
// Will's staff Id is 1000

var wilber = new Teacher("Wilber", 29, 1001);
wilber.getInfo();
// Wilber is 29 years old!
wilber.getId();
// Wilber's staff Id is 1001
```

代码中有一处地方需要注意，给子类添加**"getId"**方法的代码（**"Teacher.prototype.getId"**）一定要放在**"inheritPrototype()"**函数调用之后，因为在**“inheritPrototype()”**函数中会重写**“Teacher”**的原型。

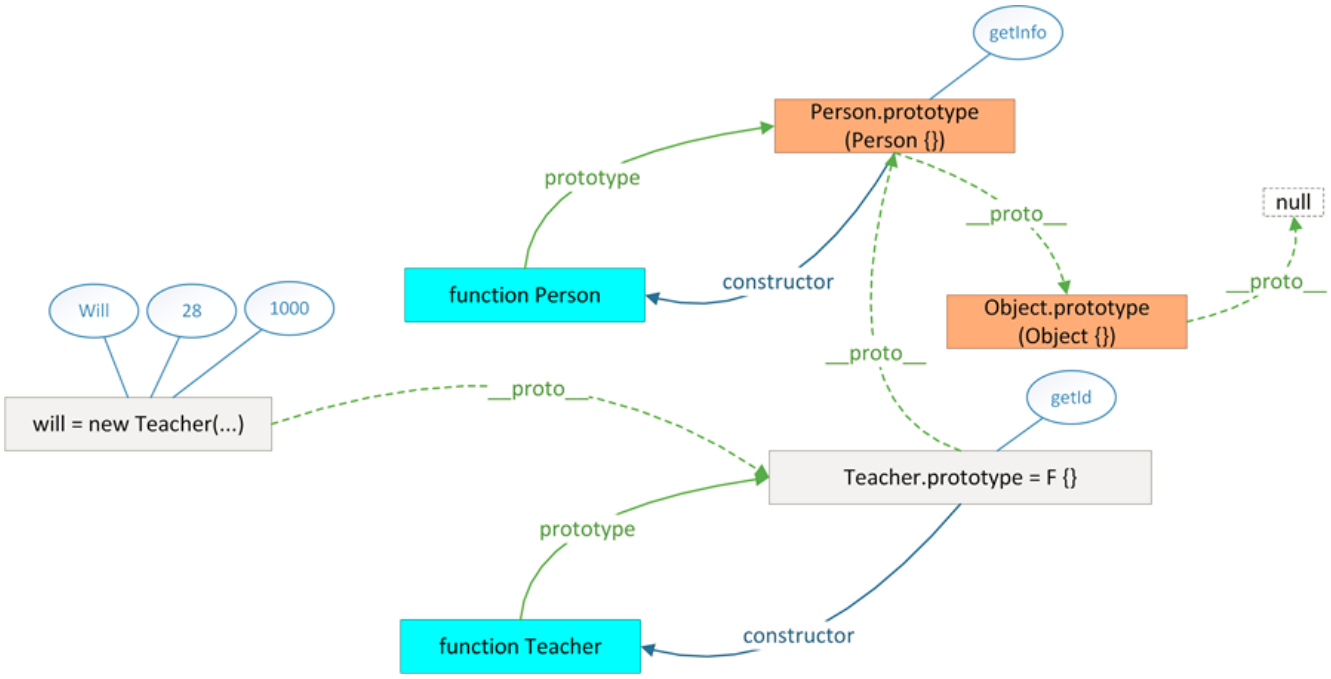
下面继续查看一下对象"will"的原型和"constructor"属性。

```
> will.__proto__
< ▼F {}
    ▶constructor: function Teacher(name, age, staffId)
    ▶getId: function ()
    ▼__proto__: Person
      ▶constructor: function Person(name, age)
      ▶getInfo: function ()
      ▶__proto__: Object
> will.__proto__.constructor
< function Teacher(name, age, staffId){
      Person.call(this, name, age)
      this.staffId = staffId;
  }
> will
< ▼Teacher {name: "Will", age: 28, staffId: 1000}
    age: 28
    name: "Will"
    staffId: 1000
    ▶__proto__: F
```

这个示例中的" inheritPrototype()"函数实现了寄生组合式继承的最简单形式。这个函数接收两个参数：子类型构造函数和父类型构造函数。

在函数内部，第一步是创建超类型原型的一个副本。第二步是为创建的副本添加 "constructor" 属性，从而弥补因重写原型而失去的默认的 "constructor" 属性。最后一步，将新创建的对象（即副本）赋值给子类型的原型。这样，我们就可以用调用 "inheritPrototype()"函数的语句，去替换前面例子中为子类型原型赋值的语句了（"Teacher.prototype = new Person();"）。

对于这个寄生组合式继承的例子，对象关系图如下：




## 总结

本文介绍了JavaScirpt中的 几种常用继承方式，我们可以通过构造函数实现继承，也可以直接基于现有的对象来实现继承。

无论哪种继承的实现，本质上都是通过JavaScript中的原型特性，结合原型链的搜索实现继承。

与其说"JavaScript是一种面向对象的语言"，更恰当的可以说"JavaScript是一种基于对象的语言"。

通过了这些介绍，相信你一定对JavaScript的继承有了一个比较清楚的认识了。