javascript学习--2017-11-15：date & 对象

笔记本：	javascript		
创建时间：	2017/11/15 21:24	**更新时间：**	2017/11/18 9:36
作者：	xiethan		
URL：	file:///C:/Program%20Files%20(x86)/Evernote/Evernote/NodeWebKit/present/index.html		

Date 类型

```
//Date 类型
var now = new Date();
console.log(now);//Wed Nov 15 2017 16:52:52 GMT+0800 (中国标准时间)
now = new Date(Date.parse("11/15/2017"));//除了这种格式还有：January 12, 2017
console.log(now);//Wed Nov 15 2017 00:00:00 GMT+0800 (中国标准时间)
now = new Date(Date.parse("Date.UTC(2015,10,5,17,55,55)"));//表示2015年11月5日下午5h 55min 55s
//利用date的构造函数
var now2 =new Date(2000,0);
console.log(now2);//Sat Jan 01 2000 00:00:00 GMT+0800 (中国标准时间)
var now2 =new Date(2015,10,5,17,55,55);
console.log(now2);//Thu Nov 05 2015 17:55:55 GMT+0800 (中国标准时间)

//记录运行时间
var start = Date.now();
//需测试的代码
//取得停止的时间
var stop=Date.now();
var result = stop - start;
//date 也继承了date的函数
// toLocalString toString valueOf
```

对象

1.属性类型

//对象 属性类型

configurable:表示能否delete删除属性而重新定义属性，能否修改属性的特性，或者能否把属性修改为访问器的属性

enumerable: 表示能否通过for-in 循环返回属性，这个特性的默认值为：undefined

writable:能否修改属性的值

value:这个属性的属性值

```
var person={
};
Object.defineProperty(person,"name",{
writable:false,
value:"ethan"
});
alert(person.name);//"ethan"
person.name="xie";
console.log(person.name);//"ethan"
```

2.访问器属性

```
var book={
_year:2004,
edition:1
}
Object.defineProperty(book, "year", {
get:function(){
return this._year;
},
set:function(newyear){
if(newyear>2004){
this._year=newyear;
this.edition=newyear-2004;
}
}
})
book.year=2015;
console.log(book.edition);//11
```

//利用上面的方式，定义多个属性，可以使用

```
Object.defineProperties(book,{
_year:{
```

```
value:2017
},
edition:{
value:1
},
year:{
get:function(){
return this._year;
},
set:function(newyear){
if(newyear>2004){
this._year=newyear;
this.edition=newyear-2004;
}
}
}
})
```

//读取属性的特性

```
var descriptor=Object.getOwnPropertyDescriptor(book,"_year");
console.log(descriptor.value);//2017
console.log(descriptor.configurable);//false
console.log(typeOf descriptor.get);//function   undefined
```

创建对象

1.工厂模式

```
//工厂模式
function createPerson(name,age,job){
var o = new Object;
o.name=name;
o.age=age;
o.job=job;
o.sayname=function(){
console.log(this.name);
}
return o;
}
var person1=createPerson("than",21,"software engineer");
var person2=createPerson("than2",21,"software engineer");
```

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问题（即怎么知道一个对象的类型），随着javascript的发展，有一个新模式出现了。
运用到的地方：特殊情况下创建函数：假设我们想创建一个具有额外方法的特殊数组，由于不能修改array的构造函数，因此可以使用这个工厂模式

1.构造函数的模式

```
//构造函数的模式
function Person(name,age,job){
this.name=name;
this.age=age;
this.job=job;
this.sayname=function(){
console.log(this.name);
}
}
var person1=new Person("than",21,"software engineer");
var person2=new Person("than2",21,"software engineer");
```

```
var person1=new Person("than",21,"software engineer");
var person2=new Person("than2",21,"software engineer");
var test=function(){
return Person("than",21,"software engineer");
}
console.log(test());//undefined
console.log(person1.constructor == Person);//下面全是true
console.log(person2.constructor == Person);
console.log(person1 instanceof Object);
console.log(person2 instanceof Person);

console.log(person1.sayname==person2.sayname);  //false
```

console.log(person1.sayname==person2.sayname);//false
；每个Person实例都包含一个function实例。说明白些，以这种方式创建函数，会导致不同的作用域链，和标识符解析。不同实例上的同名函数是不一致的。

构造函数模式，与工厂模式相比，还存在不同之处：

- 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

要创建一个新实例，必须使用new 操作符（不然返回underfined），这种方式会经历以下步骤：
创建一个新对象
将构造函数的作用域赋给新对象
执行函数中的代码
返回新对象

上面两个对象都有一个构造函数，constructor（构造函数）属性，都指向Person

```
创建自定义的构造函数意味着将来，可以将它的实例标识成为一种特定的类型，这正是构造函数模式胜过工厂模式的地方
```

把构造函数，当作函数

```
person2.sayname();//ethan2
//当作普通函数调用;
Person("window",21,"software engineer");
window.sayname();//window

//给另一个作用域使用。
var o=new Object();
console.log(Person.call(o,"object",21,"software engineer"));
o.sayname();//object
```

可以使用call(apply)在某个特殊对象的作用域中调用Person()对象，上面是在对象o的作用域中调用的。因此调用后o就拥有了所有属性与方法

问题：不同实例上的同名函数是不相等的。ES中的函数是对象，引用传递。
解决办法：这个共享了一个方法，内部只是复制指针

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


function sayname(){
alert(this.name);
}


var person1=new Person("than",21,"software engineer");
var person2=new Person("than2",21,"software engineer");
```

这样新问题又来了：在全局作用域中定义的函数，只能被某个对象调用，这让全局作用域有点名副其实，于是我们这个自定义引用类型就丝毫没有封装性可言。这些

问题可以用原型模式替代：

```
//原型模式
function human(){}
human.prototype.name="ethan";
human.prototype.age="21";
human.prototype.sayname=function(){
console.log("is prototype");
}
```

```
var human1=new human();
var human2=new human();
human1.age="22";
human1.sayname=function(){
console.log("is edited");
}
console.log(human1.age);//21  //但要读取他时，会在这个实例上搜索一个名为：name值，这个值确实存在，就返回它的值，就不必搜索原型了
console.log(human2.age);//22
console.log(human1.sayname());//is edit --- udefined
console.log(human2.sayname());//is prototype---undefined
console.log(human1.sayname == human1.sayname);//true
//true  同一个原型的同名函数才会为真
```

console.log(Person.prototype.isPrototypeOf(person1));
判断prototype，如果person1都有一个指向Person.prototype的指针，因此都返回true，ES5增加了一个方法，Object.getPrototypeOf()
证实：

```
delete human1.age;
console.log(human1.age);//21
```

判断语句（前提要肯定那个属性存在，可以用console来证实）

```
console.log(human1.hasOwnProperty("name"));//= 来自实例false 留在原型true
```

还有一个函数就是：console.log("name" in person1);
获得所有可枚举的实例属性：

```
var p1=new Person();
p1.name="ethan";
p1.age=31;
var p1key=Object.keys(p1);
console.log(p1key);//"name,age"
```

如果想要获取所以实例属性：console.log(Object.getOwnPropertyNames(Person.prototype);//constructor ……

实例中的指针 仅指向原型，而不指向构造函数

//所有原生引用模型（object array string）；（不建议：命名冲突，重写）

```
String.prototype.startsWith=function(text){
return this.indexOf(text)==0;
};
var msg="hello";
console.log(msg.startsWith("hello"));true
```

组合使用构造函数和原型模式

```
//5. 原型构造组合模式，
function Person (name, age) {
this.name = name;
this.age = age;
}
Person.prototype = {
hobby: ['running','football'],
sayName: function () { alert(this.name); },
sayAge: function () { alert(this.age); }
};
var p1 = new Person('Jack', 20);
//p1:'Jack',20; __proto__: ['running','football'],sayName,sayAge
var p2 = new Person('Mark', 18);
//p1:'Mark',18;__proto__: ['running','football'],sayName,sayAge
```

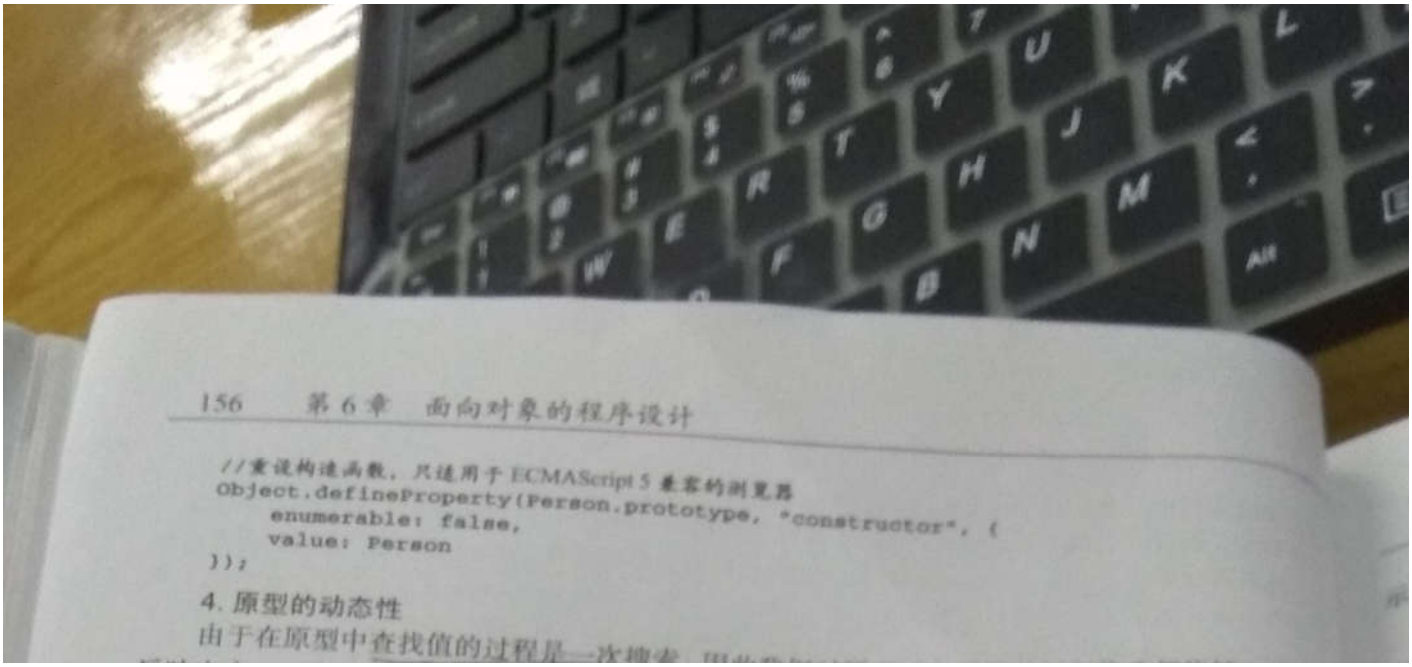
独立的属性方法放入构造函数中，而可以共享的部分则放入原型中，这样做可以最大限度节省内存而又保留对象实例的独立性，认同度最高，这是用来定义引用类型的一种默认模式。

这笔记的代码

```
<script type="text/javascript">
//创建函数

//工厂模式
function createPerson(name,age,job){
var o = new Object;
o.name=name;
o.age=age;
o.job=job;
o.sayname=function(){
console.log(this.name);
}
return o;
}
var person3=createPerson("than",21,"software engineer");
var person4=createPerson("than2",21,"software engineer");
//构造函数的模式
function Person(name,age,job){
this.name=name;
this.age=age;
this.job=job;
this.sayname=function(){
console.log(this.name);
}
}
var person1=new Person("than",21,"software engineer");
var person2=new Person("than2",21,"software engineer");
var test=function(){
return Person("than",21,"software engineer");
}
console.log(test());
console.log(person1.sayname);
console.log(person1.sayname);
console.log(person1.sayname==person2.sayname);
console.log(person1.constructor == Person);
console.log(person2.constructor == Person);
console.log(person1 instanceof Object);
console.log(person2 instanceof Person);

//把构造函数，当作函数
person2.sayname();//ethan2
//当作普通函数调用：
Person("window",21,"software engineer");
window.sayname();//window

//给另一个作用域使用。
var o=new Object();
console.log(Person.call(o,"object",21,"software engineer"));
o.sayname();

//原型模式
function human(){caree:"studer";}
human.prototype.name="ethan";
human.prototype.age="21";
human.prototype.sayname=function(){
console.log("is prototype");
}
var human1=new human();
var human2=new human();
human1.age="22";
human1.sayname=function(){
console.log("is edited");
}
console.log(human1);
console.log(human1.age);//21
delete human1.age;
console.log(human1.age);//21
console.log(human1.hasOwnProperty("name"));//= 来自实例false 留在原型true
console.log(human2.age);//22
console.log(human1.sayname());//is edit
console.log(human2.sayname());//is prototype
console.log(human1.sayname == human1.sayname);//true

console.log(human.prototype.isPrototypeOf(human1));
```

```
console.log(Object.getPrototypeOf(human1).name);

//所有原生引用模型（object array string） ;
String.prototype.startsWith=function(text){
return this.indexOf(text)==0;
};
var msg="hello";
console.log(msg.startsWith("hello"));
</script>
```

```
//es6   继承
class People{
constructor(name){
this.name=name;
}
getName(){
return this.name;
}
}
class Black extends People{
constructor(name){
super(name);
}
speak(){
return " i am black";
}
}
var B1=new Black;
console.log(B1.speak());


function Person(name, age, sal){
// 公开
this.name = name;
// 私有
var age = age;
var salary = sal;
this.abc=function(){
document.write("----<br/>"+age +"----<br/>"); //这里不能加this
}

}
var p1 = new Person('xie', 20, 10000);
document.write(p1.name +"----"+ p1.age); //xie---undefined
p1.abc();

console.log("12a"<14);//运用的是number函数转义法则 NaN与任何数比较。都是false
```



反映出来——即使是先创建了实例后修改原型也照样如此。请看下面的例子。

```
var friend = new Person();

Person.prototype.sayHi = function(){
    alert("hi");
};

friend.sayHi();    //"hi"（没有问题！）
```

PrototypePatternExample1

以上代码先创建了 Person 的一个实例，并将其保存在 person 中。然后，下一条语句在 Person prototype 中添加了一个方法 sayHi()。即使 person 实例是在添加新方法之前创建的，但它仍然可以访问这个新方法。其原因可以归结为实例与原型之间的松散连接关系。当我们调用 person.sayHi() 时，首先会在实例中搜索名为 sayHi 的属性，在没找到的情况下，会继续搜索原型。因为实例与原型之间的连接只不过是一个指针，而非一个副本，因此就可以在原型中找到新的 sayHi 属性并返回保存在那里的函数。

尽管可以随时为原型添加属性和方法，并且修改能够立即在所有对象实例中反映出来，但如果是重写整个原型对象，那么情况就不一样了。我们知道，调用构造函数时会为实例添加一个指向最初原型的 [[Prototype]]指针，而把原型修改为另外一个对象就等于切断了构造函数与最初原型之间的联系。请记住：实例中的指针仅指向原型，而不指向构造函数。看下面的例子。

```
function Person(){
}

var friend = new Person();

Person.prototype = {
    constructor: Person,
    name : "Nicholas",
    age : 29,
    job : "Software Engineer",
    sayName : function () {
        alert(this.name);
    }
};

friend.sayName();    //error
```

PrototypePatternExample1

在这个例子中，我们先创建了 Person 的一个实例，然后又重写了其原型对象。然后在调用 friend.sayName()时发生了错误，因为 friend 指向的原型中不包含以该名字命名的属性。图

```
person2.sayName();    //"Nicholas"
alert(person1.sayName == person2.sayName);  //true
```

在此，我们将 sayName()方法和所有属性直接添加到了 Person 的 prototype 属性中，构造函数变成了空函数。即使如此，也仍然可以通过调用构造函数来创建新对象，而且新对象还会具有相同的属性和方法。但与构造函数模式不同的是，新对象的这些属性和方法是由所有实例共享的。换句话说，person1 和 person2 访问的都是同一组属性和同一个 sayName()函数。要理解原型模式的工作原理必须先理解 ECMAScript 中原型对象的性质。

1. 理解原型对象

无论什么时候，只要创建了一个新函数，就会根据一组特定的规则为该函数创建一个 prototype 属性，这个属性指向函数的原型对象。在默认情况下，所有原型对象都会自动获得一个 constructor（构造函数）属性，这个属性包含一个指向 prototype 属性所在函数的指针。就拿前面的例子来说，Person.prototype.constructor 指向 Person。而通过这个构造函数，我们还可继续为原型对象添加其他属性和方法。

创建了自定义的构造函数之后，其原型对象默认只会取得 constructor 属性；至于其他方法，都是从 Object 继承而来的。当调用构造函数创建一个新实例后，该实例的内部将包含一个指针（内部属性），指向构造函数的原型对象。ECMA-262 第 5 版中管这个指针叫[[Prototype]]。虽然在脚本中没有标准的方式访问[[Prototype]]，但 Firefox、Safari 和 Chrome 在每个对象上都支持一个属性__proto__；而在其他实现中，这个属性对脚本则是完全不可见的。不过，要明确的真正重要的一点是，这个连接存在于实例与构造函数的原型对象之间，而不是存在于实例与构造函数之间。

以前面使用 Person 构造函数和 Person.prototype 创建实例的代码为例，图 6-1 展示了各个对象之间的关系。

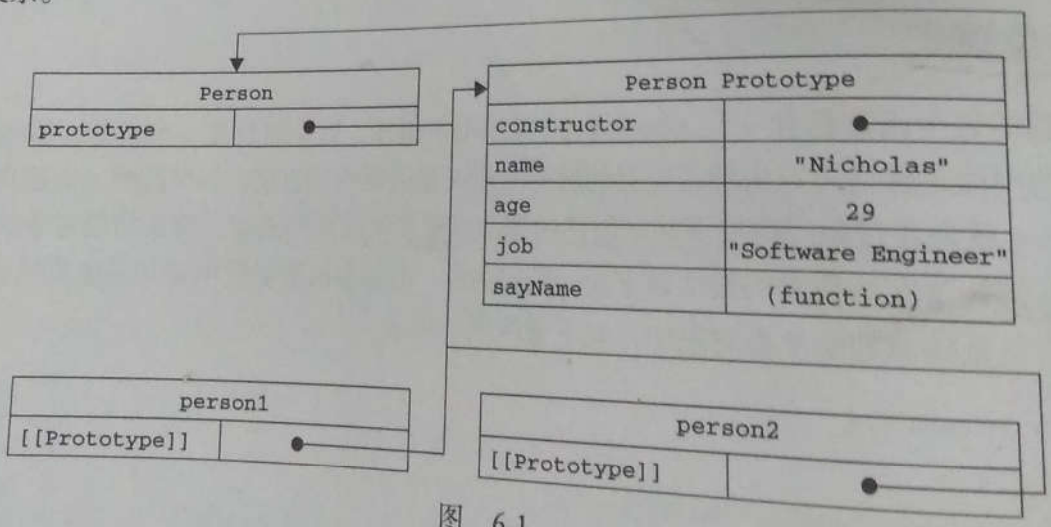
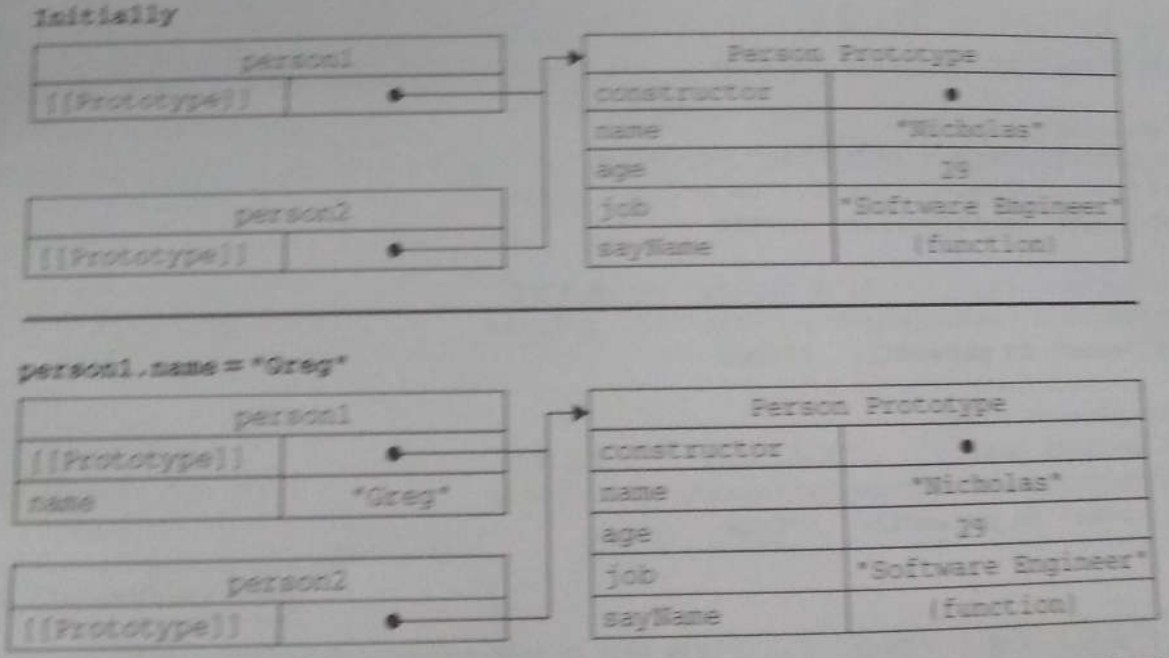


图 6-1

图 6-1 展示了 Person 构造函数、Person 的原型属性以及 Person 现有的两个实例之间的关系。在此，Person.prototype 指向了原型对象，而 Person.prototype.constructor 又指回了 Person。原型对象中除了包含 constructor 属性之外，还包括后来添加的其他属性。Person 的每个实例——person1 和 person2 都包含一个内部属性，该属性仅仅指向了 Person.prototype；换句话说，它们与构造函数没有直接的关系。此外，要格外注意的是，虽然这两个实例都不包含属性和方法，但我

```
    alert(this.name);
};

var person1 = new Person();
var person2 = new Person();

alert(person1.hasOwnProperty("name"));  //false

person1.name = "Greg";
alert(person1.name);        //"Greg"——来自实例
alert(person1.hasOwnProperty("name"));  //true

alert(person2.name);        //"Nicholas"——来自原型
alert(person2.hasOwnProperty("name"));  //false

delete person1.name;
alert(person1.name);        //"Nicholas"——来自原型
alert(person1.hasOwnProperty("name"));  //false
```

通过使用 hasOwnProperty()方法，什么时候访问的是实例属性，什么时候访问的是原型属性就一清二楚了。调用 person1.hasOwnProperty("name")时，只有当 person1 重写 name 属性后才会返回 true，因为只有这时候 name 才是一个实例属性，而非原型属性。图 6-2 展示了上面例子在不同情况下的实现与原型的关系（为了简单起见，图中省略了与 Person 构造函数的关系）。



```
delete person1.name
```

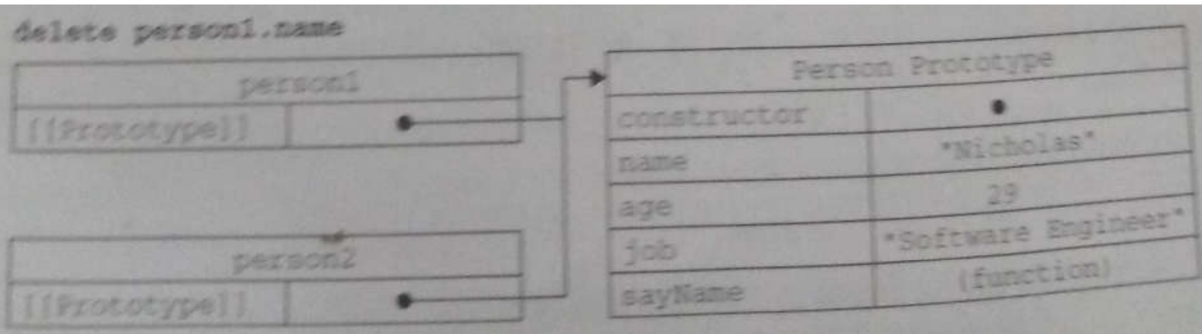


图 6-2

个过程的内幕。

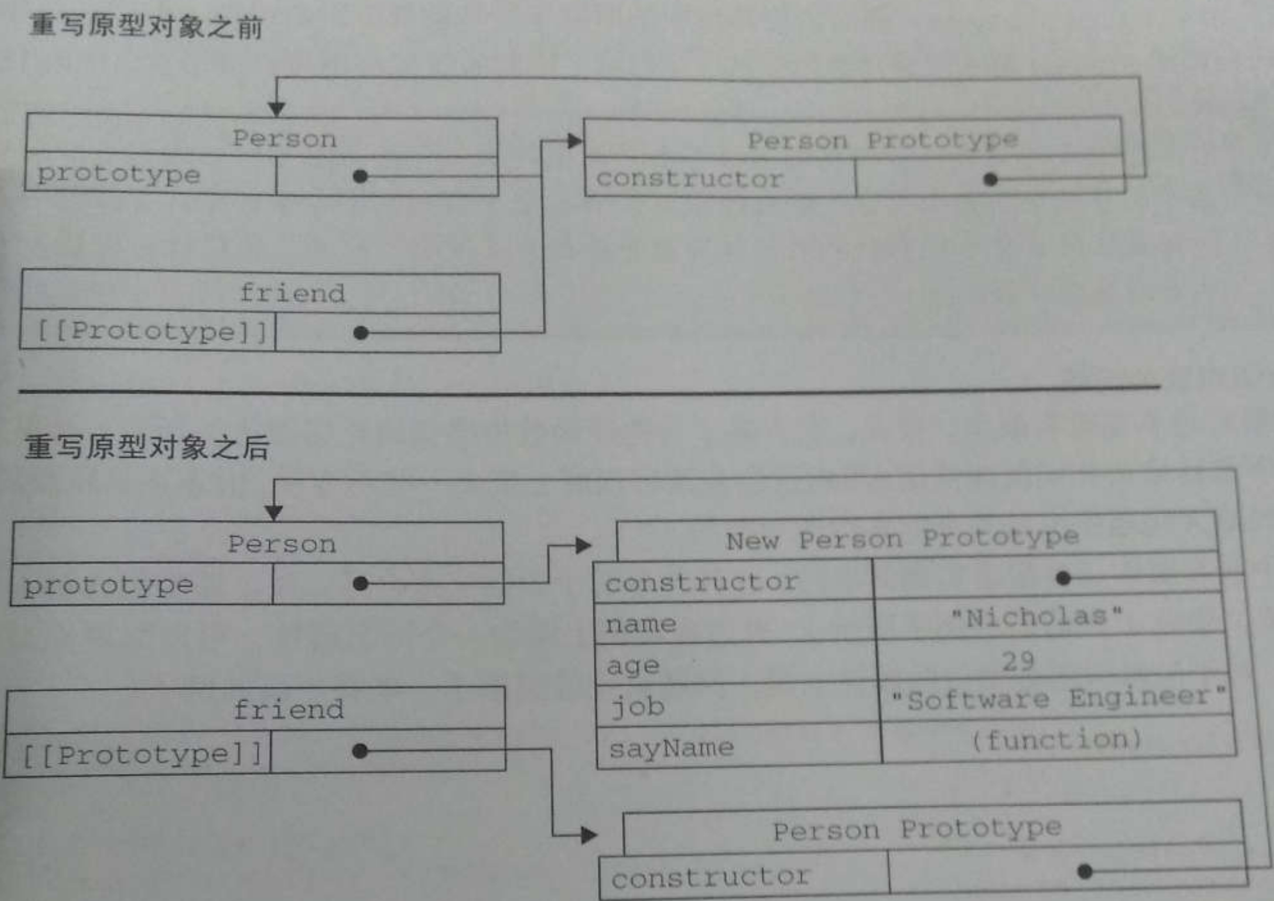


图 6-3

6-3 可以看出，重写原型对象切断了现有原型与任何之前已经存在的对象实例之间

乃然是最初的原型。

生对象的原型

模式的重要性不仅体现在创建自定义类型方面，就连所有原生的引用类型，都是采用这

所有原生引用类型（Object、Array、String，等等）都在其构造函数的原型上定义

Array.prototype 中可以找到 sort()方法，而在 String.prototype 中

g()方法，如下所示。

```
(typeof Array.prototype.sort);                    //"function"
(typeof String.prototype.substring);              //"function"
```

生对象的原型，不仅可以取得所有默认方法的引用，而且也可以定义新方法。可以

原型一样修改原生对象的原型，因此可以随时添加方法。下面的代码就给基本

加了一个名为 startsWith()的方法。

```
.prototype.startsWith = function (text) {
turn this.indexOf(text) == 0;


"Hello world!";
```