方法。之所以会这样，主要原因是实例与原型之间松散的联系。在调用 friend.sayHi()时，首先会从这个实例中搜索名为 sayHi 的属性。在没有找到的情况下，运行时会继续搜索原型对象。因为实例和原型之间的链接就是简单的指针，而不是保存的副本，所以会在原型上找到 sayHi 属性并返回这个属性保存的函数。

虽然随时能给原型添加属性和方法，并能够立即反映在所有对象实例上，但这跟重写整个原型是两回事。实例的[[Prototype]]指针是在调用构造函数时自动赋值的，这个指针即使把原型修改为不同的对象也不会变。重写整个原型会切断最初原型与构造函数的联系，但实例引用的仍然是最初的原型。记住，实例只有指向原型的指针，没有指向构造函数的指针。来看下面的例子：

```
function Person() {}

let friend = new Person();
Person.prototype = {
  constructor: Person,
  name: "Nicholas",
  age: 29,
  job: "Software Engineer",
  sayName() {
    console.log(this.name);
  }
};

friend.sayName();  // 错误
```

在这个例子中，Person 的新实例是在重写原型对象之前创建的。在调用 friend.sayName()的时候，会导致错误。这是因为 firend 指向的原型还是最初的原型，而这个原型上并没有 sayName 属性。图 8-3 展示了这里面的原因。

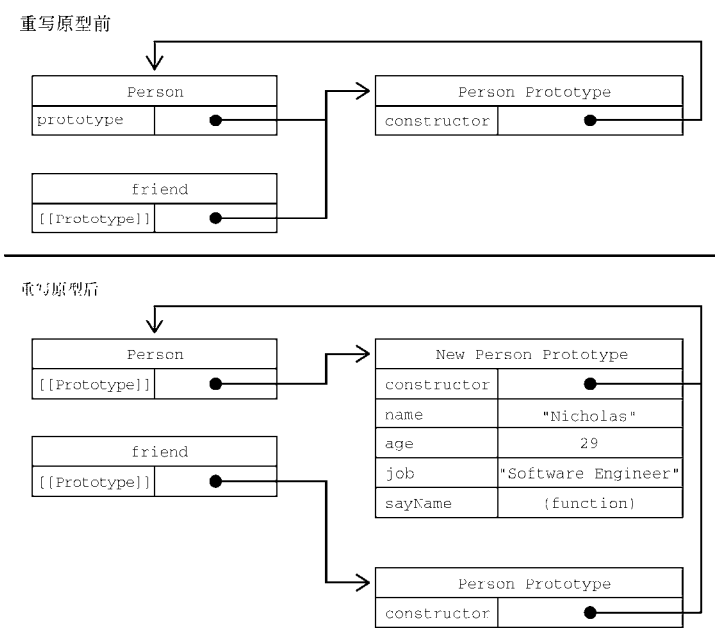

图　8-3

重写构造函数上的原型之后再创建的实例才会引用新的原型。而在此之前创建的实例仍然会引用最初的原型。

3. 原生对象原型

原型模式之所以重要，不仅体现在自定义类型上，而且还因为它也是实现所有原生引用类型的模式。所有原生引用类型的构造函数（包括 Object、Array、String 等）都在原型上定义了实例方法。比如，数组实例的 sort()方法就是 Array.prototype 上定义的，而字符串包装对象的 substring()方法也是在 String.prototype 上定义的，如下所示：

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

通过原生对象的原型可以取得所有默认方法的引用，也可以给原生类型的实例定义新的方法。可以像修改自定义对象原型一样修改原生对象原型，因此随时可以添加方法。比如，下面的代码就给 String 原始值包装类型的实例添加了一个 startsWith()方法：

```
String.prototype.startsWith = function (text) {
  return this.indexOf(text) === 0;
};

let msg = "Hello world!";
console.log(msg.startsWith("Hello"));  // true
```

如果给定字符串的开头出现了调用 startsWith()方法的文本，那么该方法会返回 true。因为这个方法是被定义在 String.prototype 上，所以当前环境下所有的字符串都可以使用这个方法。msg 是个字符串，在读取它的属性时，后台会自动创建 String 的包装实例，从而找到并调用 startsWith()方法。

> **注意** 尽管可以这么做，但并不推荐在产品环境中修改原生对象原型。这样做很可能造成误会，而且可能引发命名冲突（比如一个名称在某个浏览器实现中不存在，在另一个实现中却存在）。另外还有可能意外重写原生的方法。推荐的做法是创建一个自定义的类，继承原生类型。

4. 原型的问题

原型模式也不是没有问题。首先，它弱化了向构造函数传递初始化参数的能力，会导致所有实例默认都取得相同的属性值。虽然这会带来不便，但还不是原型的最大问题。原型的最主要问题源自它的共享特性。

我们知道，原型上的所有属性是在实例间共享的，这对函数来说比较合适。另外包含原始值的属性也还好，如前面例子中所示，可以通过在实例上添加同名属性来简单地遮蔽原型上的属性。真正的问题来自包含引用值的属性。来看下面的例子：

```
function Person() {}

Person.prototype = {
  constructor: Person,
  name: "Nicholas",
  age: 29,
  job: "Software Engineer",
  friends: ["Shelby", "Court"],
```

```
  sayName() {
    console.log(this.name);
  }
};

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

person1.friends.push("Van");

console.log(person1.friends);  // "Shelby,Court,Van"
console.log(person2.friends);  // "Shelby,Court,Van"
console.log(person1.friends === person2.friends);  // true
```

这里，Person.prototype 有一个名为 friends 的属性，它包含一个字符串数组。然后这里创建了两个 Person 的实例。person1.friends 通过 push 方法向数组中添加了一个字符串。由于这个 friends 属性存在于 Person.prototype 而非 person1 上，新加的这个字符串也会在（指向同一个数组的）person2.friends 上反映出来。如果这是有意在多个实例间共享数组，那没什么问题。但一般来说，不同的实例应该有属于自己的属性副本。这就是实际开发中通常不单独使用原型模式的原因。

8.3　继承

继承是面向对象编程中讨论最多的话题。很多面向对象语言都支持两种继承：接口继承和实现继承。前者只继承方法签名，后者继承实际的方法。接口继承在 ECMAScript 中是不可能的，因为函数没有签名。实现继承是 ECMAScript 唯一支持的继承方式，而这主要是通过原型链实现的。

8.3.1　原型链

ECMA-262 把**原型链**定义为 ECMAScript 的主要继承方式。其基本思想就是通过原型继承多个引用类型的属性和方法。重温一下构造函数、原型和实例的关系：每个构造函数都有一个原型对象，原型有一个属性指回构造函数，而实例有一个内部指针指向原型。如果原型是另一个类型的实例呢？那就意味着这个原型本身有一个内部指针指向另一个原型，相应地另一个原型也有一个指针指向另一个构造函数。这样就在实例和原型之间构造了一条原型链。这就是原型链的基本构想。

实现原型链涉及如下代码模式：

```
function SuperType() {
  this.property = true;
}

SuperType.prototype.getSuperValue = function() {
  return this.property;
};

function SubType() {
  this.subproperty = false;
}

// 继承 SuperType
SubType.prototype = new SuperType();

SubType.prototype.getSubValue = function () {
```

```
    return this.subproperty;
};

let instance = new SubType();
console.log(instance.getSuperValue()); // true
```

以上代码定义了两个类型：SuperType 和 SubType。这两个类型分别定义了一个属性和一个方法。这两个类型的主要区别是 SubType 通过创建 SuperType 的实例并将其赋值给自己的原型 SubTtype.prototype 实现了对 SuperType 的继承。这个赋值重写了 SubType 最初的原型，将其替换为 SuperType 的实例。这意味着 SuperType 实例可以访问的所有属性和方法也会存在于 SubType.prototype。这样实现继承之后，代码紧接着又给 SubType.prototype，也就是这个 SuperType 的实例添加了一个新方法。最后又创建了 SubType 的实例并调用了它继承的 getSuperValue()方法。图 8-4 展示了子类的实例与两个构造函数及其对应的原型之间的关系。

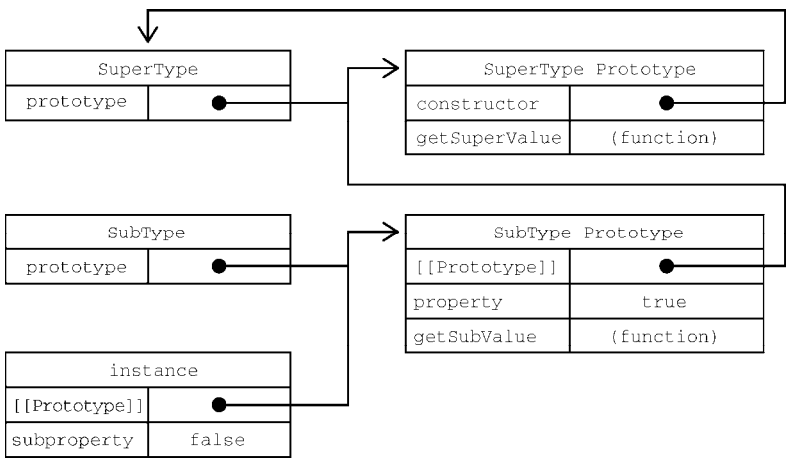

图 8-4

这个例子中实现继承的关键，是 SubType 没有使用默认原型，而是将其替换成了一个新的对象。这个新的对象恰好是 SuperType 的实例。这样一来，SubType 的实例不仅能从 SuperType 的实例中继承属性和方法，而且还与 SuperType 的原型挂上了钩。于是 instance（通过内部的[[Prototype]]）指向 SubType.prototype，而 SubType.prototype（作为 SuperType 的实例又通过内部的[[Prototype]]）指向 SuperType.prototype。注意，getSuperValue()方法还在 SuperType.prototype 对象上，而 property 属性则在 SubType.prototype 上。这是因为 getSuperValue()是一个原型方法，而 property 是一个实例属性。SubType.prototype 现在是 SuperType 的一个实例，因此 property 才会存储在它上面。还要注意，由于 SubType.prototype 的 constructor 属性被重写为指向 SuperType，所以 instance.constructor 也指向 SuperType。

原型链扩展了前面描述的原型搜索机制。我们知道，在读取实例上的属性时，首先会在实例上搜索这个属性。如果没找到，则会继承搜索实例的原型。在通过原型链实现继承之后，搜索就可以继承向上，搜索原型的原型。对前面的例子而言，调用 instance.getSuperValue()经过了 3 步搜索：instance、SubType.prototype 和 SuperType.prototype，最后一步才找到这个方法。对属性和方法的搜索会一直持续到原型链的末端。

1. 默认原型

实际上，原型链中还有一环。默认情况下，所有引用类型都继承自 Object，这也是通过原型链实现的。任何函数的默认原型都是一个 Object 的实例，这意味着这个实例有一个内部指针指向 Object.prototype。这也是为什么自定义类型能够继承包括 toString()、valueOf()在内的所有默认方法的原因。因此前面的例子还有额外一层继承关系。图 8-5 展示了完整的原型链。

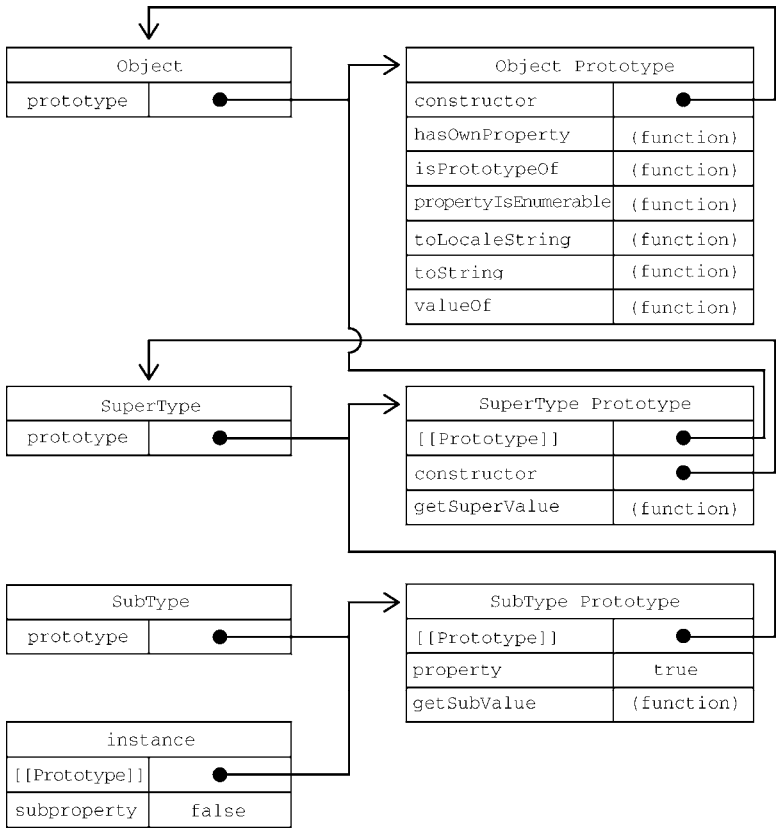

图　8-5

SubType 继承 SuperType，而 SuperType 继承 Object。在调用 instance.toString()时，实际上调用的是保存在 Object.prototype 上的方法。

2. 原型与继承关系

原型与实例的关系可以通过两种方式来确定。第一种方式是使用 instanceof 操作符，如果一个实例的原型链中出现过相应的构造函数，则 instanceof 返回 true。如下例所示：

```
console.log(instance instanceof Object);     // true
console.log(instance instanceof SuperType);  // true
console.log(instance instanceof SubType);    // true
```

从技术上讲，instance 是 Object、SuperType 和 SubType 的实例，因为 instance 的原型链中包含这些构造函数的原型。结果就是 instanceof 对所有这些构造函数都返回 true。

确定这种关系的第二种方式是使用 isPrototypeOf()方法。原型链中的每个原型都可以调用这个方法，如下例所示，只要原型链中包含这个原型，这个方法就返回 true：

```
console.log(Object.prototype.isPrototypeOf(instance));     // true
console.log(SuperType.prototype.isPrototypeOf(instance));  // true
console.log(SubType.prototype.isPrototypeOf(instance));    // true
```

3. 关于方法

子类有时候需要覆盖父类的方法，或者增加父类没有的方法。为此，这些方法必须在原型赋值之后再添加到原型上。来看下面的例子：

```
function SuperType() {
  this.property = true;
}

SuperType.prototype.getSuperValue = function() {
  return this.property;
};

function SubType() {
  this.subproperty = false;
}

// 继承 SuperType
SubType.prototype = new SuperType();

// 新方法
SubType.prototype.getSubValue = function () {
  return this.subproperty;
};

// 覆盖已有的方法
SubType.prototype.getSuperValue = function () {
  return false;
};

let instance = new SubType();
console.log(instance.getSuperValue()); // false
```

在上面的代码中，加粗的部分涉及两个方法。第一个方法 getSubValue()是 SubType 的新方法，而第二个方法 getSuperValue()是原型链上已经存在但在这里被遮蔽的方法。后面在 SubType 实例上调用 getSuperValue()时调用的是这个方法。而 SuperType 的实例仍然会调用最初的方法。重点在于上述两个方法都是在把原型赋值为 SuperType 的实例之后定义的。

另一个要理解的重点是，以对象字面量方式创建原型方法会破坏之前的原型链，因为这相当于重写了原型链。下面是一个例子：

```
function SuperType() {
  this.property = true;
}

SuperType.prototype.getSuperValue = function() {
  return this.property;
};

function SubType() {
  this.subproperty = false;
}
```

```
// 继承 SuperType
SubType.prototype = new SuperType();

// 通过对象字面量添加新方法，这会导致上一行无效
SubType.prototype = {
  getSubValue() {
    return this.subproperty;
  },

  someOtherMethod() {
    return false;
  }
};

let instance = new SubType();
console.log(instance.getSuperValue()); // 出错!
```

在这段代码中，子类的原型在被赋值为 SuperType 的实例后，又被一个对象字面量覆盖了。覆盖后的原型是一个 Object 的实例，而不再是 SuperType 的实例。因此之前的原型链就断了。SubType 和 SuperType 之间也没有关系了。

4. 原型链的问题

原型链虽然是实现继承的强大工具，但它也有问题。主要问题出现在原型中包含引用值的时候。前面在谈到原型的问题时也提到过，原型中包含的引用值会在所有实例间共享，这也是为什么属性通常会在构造函数中定义而不会定义在原型上的原因。在使用原型实现继承时，原型实际上变成了另一个类型的实例。这意味着原先的实例属性摇身一变成为了原型属性。下面的例子揭示了这个问题：

```
function SuperType() {
  this.colors = ["red", "blue", "green"];
}

function SubType() {}

// 继承 SuperType
SubType.prototype = new SuperType();

let instance1 = new SubType();
instance1.colors.push("black");
console.log(instance1.colors); // "red,blue,green,black"

let instance2 = new SubType();
console.log(instance2.colors); // "red,blue,green,black"
```

在这个例子中，SuperType 构造函数定义了一个 colors 属性，其中包含一个数组（引用值）。每个 SuperType 的实例都会有自己的 colors 属性，包含自己的数组。但是，当 SubType 通过原型继承 SuperType 后，SubType.prototype 变成了 SuperType 的一个实例，因而也获得了自己的 colors 属性。这类似于创建了 SubType.prototype.colors 属性。最终结果是，SubType 的所有实例都会共享这个 colors 属性。这一点通过 instance1.colors 上的修改也能反映到 instance2.colors 上就可以看出来。

原型链的第二个问题是，子类型在实例化时不能给父类型的构造函数传参。事实上，我们无法在不影响所有对象实例的情况下把参数传进父类的构造函数。再加上之前提到的原型中包含引用值的问题，就导致原型链基本不会被单独使用。

8.3.2 盗用构造函数

为了解决原型包含引用值导致的继承问题，一种叫作“盗用构造函数”（constructor stealing）的技术在开发社区流行起来（这种技术有时也称作“对象伪装”或“经典继承”）。基本思路很简单：在子类构造函数中调用父类构造函数。因为毕竟函数就是在特定上下文中执行代码的简单对象，所以可以使用apply()和call()方法以新创建的对象为上下文执行构造函数。来看下面的例子：

```
function SuperType() {
  this.colors = ["red", "blue", "green"];
}

function SubType() {
  // 继承 SuperType
  SuperType.call(this);
}

let instance1 = new SubType();
instance1.colors.push("black");
console.log(instance1.colors); // "red,blue,green,black"

let instance2 = new SubType();
console.log(instance2.colors); // "red,blue,green"
```

示例中加粗的代码展示了盗用构造函数的调用。通过使用call()（或apply()）方法，SuperType构造函数在为SubType的实例创建的新对象的上下文中执行了。这相当于新的SubType对象上运行了SuperType()函数中的所有初始化代码。结果就是每个实例都会有自己的colors属性。

1. 传递参数

相比于使用原型链，盗用构造函数的一个优点就是可以在子类构造函数中向父类构造函数传参。来看下面的例子：

```
function SuperType(name){
  this.name = name;
}

function SubType() {
  // 继承 SuperType 并传参
  SuperType.call(this, "Nicholas");

  // 实例属性
  this.age = 29;
}

let instance = new SubType();
console.log(instance.name); // "Nicholas";
console.log(instance.age);  // 29
```

在这个例子中，SuperType构造函数接收一个参数name，然后将它赋值给一个属性。在SubType构造函数中调用SuperType构造函数时传入这个参数，实际上会在SubType的实例上定义name属性。为确保SuperType构造函数不会覆盖SubType定义的属性，可以在调用父类构造函数之后再给子类实例添加额外的属性。

2. 盗用构造函数的问题

盗用构造函数的主要缺点，也是使用构造函数模式自定义类型的问题：必须在构造函数中定义方法，

因此函数不能重用。此外，子类也不能访问父类原型上定义的方法，因此所有类型只能使用构造函数模式。由于存在这些问题，盗用构造函数基本上也不能单独使用。

8.3.3　组合继承

组合继承（有时候也叫伪经典继承）综合了原型链和盗用构造函数，将两者的优点集中了起来。基本的思路是使用原型链继承原型上的属性和方法，而通过盗用构造函数继承实例属性。这样既可以把方法定义在原型上以实现重用，又可以让每个实例都有自己的属性。来看下面的例子：

```
function SuperType(name){
  this.name = name;
  this.colors = ["red", "blue", "green"];
}

SuperType.prototype.sayName = function() {
  console.log(this.name);
};

function SubType(name, age){
  // 继承属性
  SuperType.call(this, name);

  this.age = age;
}

// 继承方法
SubType.prototype = new SuperType();

SubType.prototype.sayAge = function() {
  console.log(this.age);
};

let instance1 = new SubType("Nicholas", 29);
instance1.colors.push("black");
console.log(instance1.colors);  // "red,blue,green,black"
instance1.sayName();            // "Nicholas";
instance1.sayAge();             // 29

let instance2 = new SubType("Greg", 27);
console.log(instance2.colors);  // "red,blue,green"
instance2.sayName();            // "Greg";
instance2.sayAge();             // 27
```

在这个例子中，SuperType 构造函数定义了两个属性，name 和 colors，而它的原型上也定义了一个方法叫 sayName()。SubType 构造函数调用了 SuperType 构造函数，传入了 name 参数，然后又定义了自己的属性 age。此外，SubType.prototype 也被赋值为 SuperType 的实例。原型赋值之后，又在这个原型上添加了新方法 sayAge()。这样，就可以创建两个 SubType 实例，让这两个实例都有自己的属性，包括 colors，同时还共享相同的方法。

组合继承弥补了原型链和盗用构造函数的不足，是 JavaScript 中使用最多的继承模式。而且组合继承也保留了 instanceof 操作符和 isPrototypeOf()方法识别合成对象的能力。

8.3.4 原型式继承

2006 年，Douglas Crockford 写了一篇文章：《JavaScript 中的原型式继承》（“Prototypal Inheritance in JavaScript”）。这篇文章介绍了一种不涉及严格意义上构造函数的继承方法。他的出发点是即使不自定义类型也可以通过原型实现对象之间的信息共享。文章最终给出了一个函数：

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

这个 object()函数会创建一个临时构造函数，将传入的对象赋值给这个构造函数的原型，然后返回这个临时类型的一个实例。本质上，object()是对传入的对象执行了一次浅复制。来看下面的例子：

```
let person = {
  name: "Nicholas",
  friends: ["Shelby", "Court", "Van"]
};

let anotherPerson = object(person);
anotherPerson.name = "Greg";
anotherPerson.friends.push("Rob");

let yetAnotherPerson = object(person);
yetAnotherPerson.name = "Linda";
yetAnotherPerson.friends.push("Barbie");

console.log(person.friends);  // "Shelby,Court,Van,Rob,Barbie"
```

Crockford 推荐的原型式继承适用于这种情况：你有一个对象，想在它的基础上再创建一个新对象。你需要把这个对象先传给 object()，然后再对返回的对象进行适当修改。在这个例子中，person 对象定义了另一个对象也应该共享的信息，把它传给 object()之后会返回一个新对象。这个新对象的原型是 person，意味着它的原型上既有原始值属性又有引用值属性。这也意味着 person.friends 不仅是 person 的属性，也会跟 anotherPerson 和 yetAnotherPerson 共享。这里实际上克隆了两个 person。

ECMAScript 5 通过增加 Object.create()方法将原型式继承的概念规范化了。这个方法接收两个参数：作为新对象原型的对象，以及给新对象定义额外属性的对象（第二个可选）。在只有一个参数时，Object.create()与这里的 object()方法效果相同：

```
let person = {
  name: "Nicholas",
  friends: ["Shelby", "Court", "Van"]
};

let anotherPerson = Object.create(person);
anotherPerson.name = "Greg";
anotherPerson.friends.push("Rob");

let yetAnotherPerson = Object.create(person);
yetAnotherPerson.name = "Linda";
yetAnotherPerson.friends.push("Barbie");

console.log(person.friends);  // "Shelby,Court,Van,Rob,Barbie"
```