```
// }

/**
 * 如前所述，构造函数有一个 prototype 属性
 * 引用其原型对象，而这个原型对象也有一个
 * constructor 属性，引用这个构造函数
 * 换句话说，两者循环引用：
 */
console.log(Person.prototype.constructor === Person); // true

/**
 * 正常的原型链都会终止于 Object 的原型对象
 * Object 原型的原型是 null
 */
console.log(Person.prototype.__proto__ === Object.prototype);   // true
console.log(Person.prototype.__proto__.constructor === Object); // true
console.log(Person.prototype.__proto__.__proto__ === null);     // true

console.log(Person.prototype.__proto__);
// {
//   constructor: f Object(),
//   toString: ...
//   hasOwnProperty: ...
//   isPrototypeOf: ...
//   ...
// }


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

/**
 * 构造函数、原型对象和实例
 * 是 3 个完全不同的对象：
 */
console.log(person1 !== Person);            // true
console.log(person1 !== Person.prototype); // true
console.log(Person.prototype !== Person);  // true

/**
  * 实例通过__proto__链接到原型对象，
  * 它实际上指向隐藏特性[[Prototype]]
  *
  * 构造函数通过 prototype 属性链接到原型对象
  *
  * 实例与构造函数没有直接联系，与原型对象有直接联系
  */
console.log(person1.__proto__ === Person.prototype);   // true
conosle.log(person1.__proto__.constructor === Person); // true

/**
 * 同一个构造函数创建的两个实例
 * 共享同一个原型对象：
 */
console.log(person1.__proto__ === person2.__proto__); // true

/**
 * instanceof 检查实例的原型链中
```

```
 * 是否包含指定构造函数的原型：
 */
console.log(person1 instanceof Person);           // true
console.log(person1 instanceof Object);           // true
console.log(Person.prototype instanceof Object);  // true
```

对于前面例子中的 Person 构造函数和 Person.prototype，可以通过图 8-1 看出各个对象之间的关系。

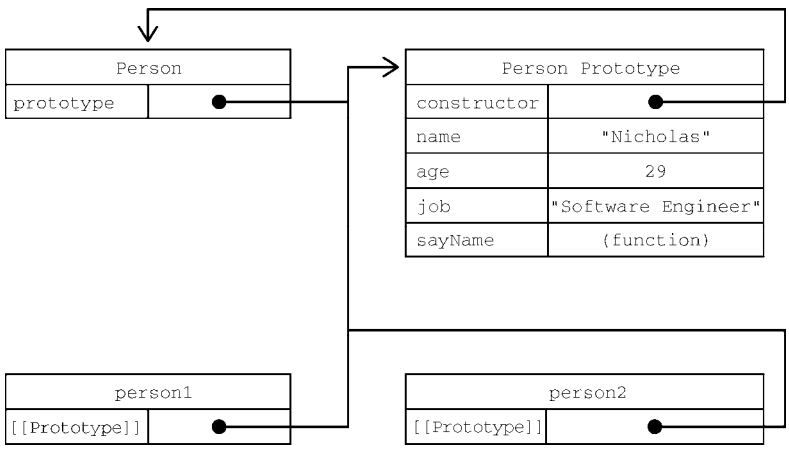

图 8-1

图 8-1 展示了 Person 构造函数、Person 的原型对象和 Person 现有两个实例之间的关系。注意，Person.prototype 指向原型对象，而 Person.prototype.contructor 指回 Person 构造函数。原型对象包含 constructor 属性和其他后来添加的属性。Person 的两个实例 person1 和 person2 都只有一个内部属性指回 Person.prototype，而且两者都与构造函数没有直接联系。另外要注意，虽然这两个实例都没有属性和方法，但 person1.sayName()可以正常调用。这是由于对象属性查找机制的原因。

虽然不是所有实现都对外暴露了[[Prototype]]，但可以使用 isPrototypeOf()方法确定两个对象之间的这种关系。本质上，isPrototypeOf()会在传入参数的[[Prototype]]指向调用它的对象时返回 true，如下所示：

```
console.log(Person.prototype.isPrototypeOf(person1));  // true
console.log(Person.prototype.isPrototypeOf(person2));  // true
```

这里通过原型对象调用 isPrototypeOf()方法检查了 person1 和 person2。因为这两个例子内部都有链接指向 Person.prototype，所以结果都返回 true。

ECMAScript 的 Object 类型有一个方法叫 Object.getPrototypeOf()，返回参数的内部特性[[Prototype]]的值。例如：

```
console.log(Object.getPrototypeOf(person1) == Person.prototype);  // true
console.log(Object.getPrototypeOf(person1).name);                 // "Nicholas"
```

第一行代码简单确认了 Object.getPrototypeOf()返回的对象就是传入对象的原型对象。第二行代码则取得了原型对象上 name 属性的值，即"Nicholas"。使用 Object.getPrototypeOf()可以方便地取得一个对象的原型，而这在通过原型实现继承时显得尤为重要（本章后面会介绍）。

Object 类型还有一个 setPrototypeOf()方法，可以向实例的私有特性[[Prototype]]写入一个新值。这样就可以重写一个对象的原型继承关系：

```
let biped = {
  numLegs: 2
};
let person = {
  name: 'Matt'
};

Object.setPrototypeOf(person, biped);

console.log(person.name);                                // Matt
console.log(person.numLegs);                             // 2
console.log(Object.getPrototypeOf(person) === biped);  // true
```

> **警告**　Object.setPrototypeOf()可能会严重影响代码性能。Mozilla 文档说得很清楚："在所有浏览器和 JavaScript 引擎中，修改继承关系的影响都是微妙且深远的。这种影响并不仅是执行 Object.setPrototypeOf()语句那么简单，而是会涉及所有访问了那些修改过[[Prototype]]的对象的代码。"

为避免使用 Object.setPrototypeOf()可能造成的性能下降，可以通过 Object.create()来创建一个新对象，同时为其指定原型：

```
let biped = {
  numLegs: 2
};
let person = Object.create(biped);
person.name = 'Matt';

console.log(person.name);                                // Matt
console.log(person.numLegs);                             // 2
console.log(Object.getPrototypeOf(person) === biped);  // true
```

2. 原型层级

在通过对象访问属性时，会按照这个属性的名称开始搜索。搜索开始于对象实例本身。如果在这个实例上发现了给定的名称，则返回该名称对应的值。如果没有找到这个属性，则搜索会沿着指针进入原型对象，然后在原型对象上找到属性后，再返回对应的值。因此，在调用 person1.sayName()时，会发生两步搜索。首先，JavaScript 引擎会问："person1 实例有 sayName 属性吗？"答案是没有。然后，继续搜索并问："person1 的原型有 sayName 属性吗？"答案是有。于是就返回了保存在原型上的这个函数。在调用 person2.sayName()时，会发生同样的搜索过程，而且也会返回相同的结果。这就是原型用于在多个对象实例间共享属性和方法的原理。

> **注意**　前面提到的 constructor 属性只存在于原型对象，因此通过实例对象也是可以访问到的。

虽然可以通过实例读取原型对象上的值，但不可能通过实例重写这些值。如果在实例上添加了一个与原型对象中同名的属性，那就会在实例上创建这个属性，这个属性会遮住原型对象上的属性。下面看一个例子：

```
function Person() {}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function() {
  console.log(this.name);
};

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

person1.name = "Greg";
console.log(person1.name);  // "Greg", 来自实例
console.log(person2.name);  // "Nicholas", 来自原型
```

在这个例子中，person1 的 name 属性遮蔽了原型对象上的同名属性。虽然 person1.name 和 person2.name 都返回了值，但前者返回的是"Greg"（来自实例），后者返回的是"Nicholas"（来自原型）。当 console.log()访问 person1.name 时，会先在实例上搜索个属性。因为这个属性在实例上存在，所以就不会再搜索原型对象了。而在访问 person2.name 时，并没有在实例上找到这个属性，所以会继续搜索原型对象并使用定义在原型上的属性。

只要给对象实例添加一个属性，这个属性就会**遮蔽**（shadow）原型对象上的同名属性，也就是虽然不会修改它，但会屏蔽对它的访问。即使在实例上把这个属性设置为 null，也不会恢复它和原型的联系。不过，使用 delete 操作符可以完全删除实例上的这个属性，从而让标识符解析过程能够继续搜索原型对象。

```
function Person() {}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function() {
  console.log(this.name);
};

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

person1.name = "Greg";
console.log(person1.name);  // "Greg", 来自实例
console.log(person2.name);  // "Nicholas", 来自原型

delete person1.name;
console.log(person1.name);  // "Nicholas", 来自原型
```

这个修改后的例子中使用 delete 删除了 person1.name，这个属性之前以"Greg"遮蔽了原型上的同名属性。然后原型上 name 属性的联系就恢复了，因此再访问 person1.name 时，就会返回原型对象上这个属性的值。

hasOwnProperty()方法用于确定某个属性是在实例上还是在原型对象上。这个方法是继承自 Object 的，会在属性存在于调用它的对象实例上时返回 true，如下面的例子所示：

```
function Person() {}

Person.prototype.name = "Nicholas";
```

```
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function() {
  console.log(this.name);
};

let person1 = new Person();
let person2 = new Person();
console.log(person1.hasOwnProperty("name")); // false

person1.name = "Greg";
console.log(person1.name); // "Greg", 来自实例
console.log(person1.hasOwnProperty("name")); // true

console.log(person2.name); // "Nicholas", 来自原型
console.log(person2.hasOwnProperty("name")); // false

delete person1.name;
console.log(person1.name); // "Nicholas", 来自原型
console.log(person1.hasOwnProperty("name")); // false
```

在这个例子中，通过调用 hasOwnProperty()能够清楚地看到访问的是实例属性还是原型属性。调用 person1.hasOwnProperty("name")只在重写 person1 上 name 属性的情况下才返回 true，表明此时 name 是一个实例属性，不是原型属性。图 8-2 形象地展示了上面例子中各个步骤的状态。（为简单起见，图中省略了 Person 构造函数。）

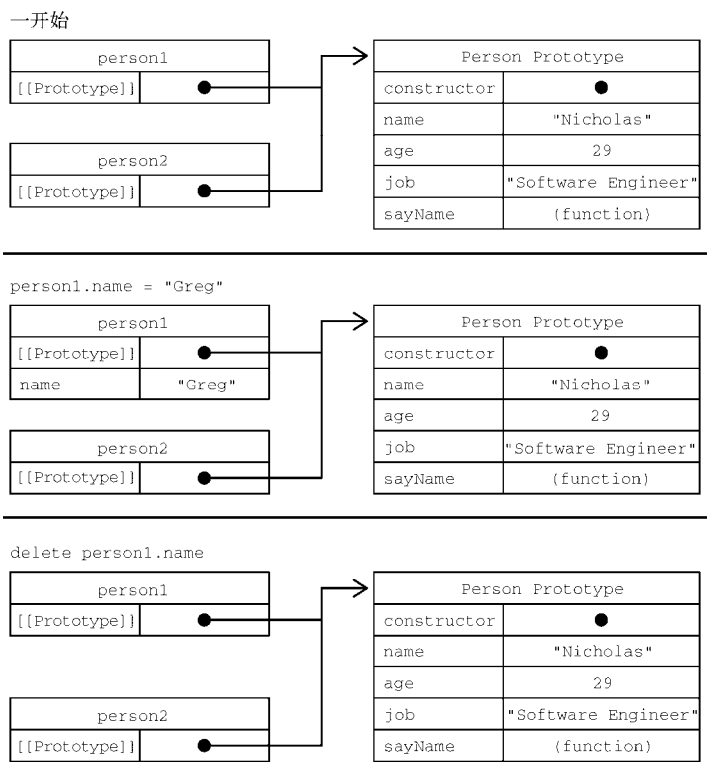

图　8-2

> **注意** ECMAScript 的 Object.getOwnPropertyDescriptor()方法只对实例属性有效。要取得原型属性的描述符，就必须直接在原型对象上调用 Object.getOwnPropertyDescriptor()。

3. 原型和 in 操作符

有两种方式使用 in 操作符：单独使用和在 for-in 循环中使用。在单独使用时，in 操作符会在可以通过对象访问指定属性时返回 true，无论该属性是在实例上还是在原型上。来看下面的例子：

```
function Person() {}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function() {
  console.log(this.name);
};

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

console.log(person1.hasOwnProperty("name")); // false
console.log("name" in person1); // true

person1.name = "Greg";
console.log(person1.name); // "Greg", 来自实例
console.log(person1.hasOwnProperty("name")); // true
console.log("name" in person1); // true

console.log(person2.name); // "Nicholas", 来自原型
console.log(person2.hasOwnProperty("name")); // false
console.log("name" in person2); // true

delete person1.name;
console.log(person1.name); // "Nicholas", 来自原型
console.log(person1.hasOwnProperty("name")); // false
console.log("name" in person1); // true
```

在上面整个例子中，name 随时可以通过实例或通过原型访问到。因此，调用"name" in persoon1 时始终返回 true，无论这个属性是否在实例上。如果要确定某个属性是否存在于原型上，则可以像下面这样同时使用 hasOwnProperty()和 in 操作符：

```
function hasPrototypeProperty(object, name){
  return !object.hasOwnProperty(name) && (name in object);
}
```

只要通过对象可以访问，in 操作符就返回 true，而 hasOwnProperty()只有属性存在于实例上时才返回 true。因此，只要 in 操作符返回 true 且 hasOwnProperty()返回 false，就说明该属性是一个原型属性。来看下面的例子：

```
function Person() {}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
```

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

let person = new Person();
console.log(hasPrototypeProperty(person, "name")); // true

person.name = "Greg";
console.log(hasPrototypeProperty(person, "name")); // false
```

在这里，name 属性首先只存在于原型上，所以 hasPrototypeProperty()返回 true。而在实例上重写这个属性后，实例上也有了这个属性，因此 hasPrototypeProperty()返回 false。即便此时原型对象还有 name 属性，但因为实例上的属性遮蔽了它，所以不会用到。

在 for-in 循环中使用 in 操作符时，可以通过对象访问且可以被枚举的属性都会返回，包括实例属性和原型属性。遮蔽原型中不可枚举（[[Enumerable]]特性被设置为 false）属性的实例属性也会在 for-in 循环中返回，因为默认情况下开发者定义的属性都是可枚举的。

要获得对象上所有可枚举的实例属性，可以使用 Object.keys()方法。这个方法接收一个对象作为参数，返回包含该对象所有可枚举属性名称的字符串数组。比如：

```
function Person() {}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function() {
  console.log(this.name);
};

let keys = Object.keys(Person.prototype);
console.log(keys);   // "name,age,job,sayName"
let p1 = new Person();
p1.name = "Rob";
p1.age = 31;
let p1keys = Object.keys(p1);
console.log(p1keys); // "[name,age]"
```

这里，keys 变量保存的数组中包含"name"、"age"、"job"和"sayName"。这是正常情况下通过 for-in 返回的顺序。而在 Person 的实例上调用时，Object.keys()返回的数组中只包含"name"和"age"两个属性。

如果想列出所有实例属性，无论是否可以枚举，都可以使用 Object.getOwnPropertyNames()：

```
let keys = Object.getOwnPropertyNames(Person.prototype);
console.log(keys);   // "[constructor,name,age,job,sayName]"
```

注意，返回的结果中包含了一个不可枚举的属性 constructor。Object.keys()和 Object.getOwnPropertyNames()在适当的时候都可用来代替 for-in 循环。

在 ECMAScript 6 新增符号类型之后，相应地出现了增加一个 Object.getOwnPropertyNames()的兄弟方法的需求，因为以符号为键的属性没有名称的概念。因此，Object.getOwnPropertySymbols()方法就出现了，这个方法与 Object.getOwnPropertyNames()类似，只是针对符号而已：

```
let k1 = Symbol('k1'),
    k2 = Symbol('k2');
```

```
let o = {
  [k1]: 'k1',
  [k2]: 'k2'
};

console.log(Object.getOwnPropertySymbols(o));
// [Symbol(k1), Symbol(k2)]
```

4. 属性枚举顺序

for-in 循环、Object.keys()、Object.getOwnPropertyNames()、Object.getOwnPropertySymbols()以及 Object.assign()在属性枚举顺序方面有很大区别。for-in 循环和 Object.keys()的枚举顺序是不确定的，取决于 JavaScript 引擎，可能因浏览器而异。

Object.getOwnPropertyNames()、Object.getOwnPropertySymbols()和 Object.assign()的枚举顺序是确定性的。先以升序枚举数值键，然后以插入顺序枚举字符串和符号键。在对象字面量中定义的键以它们逗号分隔的顺序插入。

```
let k1 = Symbol('k1'),
    k2 = Symbol('k2');

let o = {
  1: 1,
  first: 'first',
  [k1]: 'sym2',
  second: 'second',
  0: 0
};

o[k2] = 'sym2';
o[3] = 3;
o.third = 'third';
o[2] = 2;

console.log(Object.getOwnPropertyNames(o));
// ["0", "1", "2", "3", "first", "second", "third"]

console.log(Object.getOwnPropertySymbols(o));
// [Symbol(k1), Symbol(k2)]
```

8.2.5 对象迭代

在 JavaScript 有史以来的大部分时间内，迭代对象属性都是一个难题。ECMAScript 2017 新增了两个静态方法，用于将对象内容转换为序列化的——更重要的是可迭代的——格式。这两个静态方法 Object.values()和 Object.entries()接收一个对象，返回它们内容的数组。Object.values()返回对象值的数组，Object.entries()返回键/值对的数组。

下面的示例展示了这两个方法：

```
const o = {
  foo: 'bar',
  baz: 1,
  qux: {}
};

console.log(Object.values(o));
```

```
// ["bar", 1, {}]

console.log(Object.entries((o)));
// [["foo", "bar"], ["baz", 1], ["qux", {}]]
```

注意，非字符串属性会被转换为字符串输出。另外，这两个方法执行对象的浅复制：

```
const o = {
  qux: {}
};

console.log(Object.values(o)[0] === o.qux);
// true

console.log(Object.entries(o)[0][1] === o.qux);
// true
```

符号属性会被忽略：

```
const sym = Symbol();
const o = {
  [sym]: 'foo'
};

console.log(Object.values(o));
// []

console.log(Object.entries((o)));
// []
```

1. 其他原型语法

有读者可能注意到了，在前面的例子中，每次定义一个属性或方法都会把 Person.prototype 重写一遍。为了减少代码冗余，也为了从视觉上更好地封装原型功能，直接通过一个包含所有属性和方法的对象字面量来重写原型成为了一种常见的做法，如下面的例子所示：

```
function Person() {}

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

在这个例子中，Person.prototype 被设置为等于一个通过对象字面量创建的新对象。最终结果是一样的，只有一个问题：这样重写之后，Person.prototype 的 constructor 属性就不指向 Person 了。在创建函数时，也会创建它的 prototype 对象，同时会自动给这个原型的 constructor 属性赋值。而上面的写法完全重写了默认的 prototype 对象，因此其 constructor 属性也指向了完全不同的新对象（Object 构造函数），不再指向原来的构造函数。虽然 instanceof 操作符还能可靠地返回值，但我们不能再依靠 constructor 属性来识别类型了，如下面的例子所示：

```
let friend = new Person();

console.log(friend instanceof Object);      // true
console.log(friend instanceof Person);      // true
```

```
console.log(friend.constructor == Person);  // false
console.log(friend.constructor == Object);  // true
```

这里，instanceof 仍然对 Object 和 Person 都返回 true。但 constructor 属性现在等于 Object 而不是 Person 了。如果 constructor 的值很重要，则可以像下面这样在重写原型对象时专门设置一下它的值：

```
function Person() {
}

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

这次的代码中特意包含了 constructor 属性，并将它设置为 Person，保证了这个属性仍然包含恰当的值。

但要注意，以这种方式恢复 constructor 属性会创建一个[[Enumerable]]为 true 的属性。而原生 constructor 属性默认是不可枚举的。因此，如果你使用的是兼容 ECMAScript 的 JavaScript 引擎，那可能会改为使用 Object.defineProperty()方法来定义 constructor 属性：

```
function Person() {}

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

// 恢复 constructor 属性
Object.defineProperty(Person.prototype, "constructor", {
  enumerable: false,
  value: Person
});
```

2. 原型的动态性

因为从原型上搜索值的过程是动态的，所以即使实例在修改原型之前已经存在，任何时候对原型对象所做的修改也会在实例上反映出来。下面是一个例子：

```
let friend = new Person();

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

friend.sayHi();   // "hi", 没问题!
```

以上代码先创建一个 Person 实例并保存在 friend 中。然后一条语句在 Person.prototype 上添加了一个名为 sayHi()的方法。虽然 friend 实例是在添加方法之前创建的，但它仍然可以访问这个