Object.create()的第二个参数与 Object.defineProperties()的第二个参数一样：每个新增属性都通过各自的描述符来描述。以这种方式添加的属性会遮蔽原型对象上的同名属性。比如：

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

let anotherPerson = Object.create(person, {
  name: {
    value: "Greg"
  }
});
console.log(anotherPerson.name);  // "Greg"
```

原型式继承非常适合不需要单独创建构造函数，但仍然需要在对象间共享信息的场合。但要记住，属性中包含的引用值始终会在相关对象间共享，跟使用原型模式是一样的。

8.3.5　寄生式继承

与原型式继承比较接近的一种继承方式是**寄生式继承**（parasitic inheritance），也是 Crockford 首倡的一种模式。寄生式继承背后的思路类似于寄生构造函数和工厂模式：创建一个实现继承的函数，以某种方式增强对象，然后返回这个对象。基本的寄生继承模式如下：

```
function createAnother(original){
  let clone = object(original);  // 通过调用函数创建一个新对象
  clone.sayHi = function() {     // 以某种方式增强这个对象
    console.log("hi");
  };
  return clone;            // 返回这个对象
}
```

在这段代码中，createAnother()函数接收一个参数，就是新对象的基准对象。这个对象 original 会被传给 object()函数，然后将返回的新对象赋值给 clone。接着给 clone 对象添加一个新方法 sayHi()。最后返回这个对象。可以像下面这样使用 createAnother()函数：

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

let anotherPerson = createAnother(person);
anotherPerson.sayHi();  // "hi"
```

这个例子基于 person 对象返回了一个新对象。新返回的 anotherPerson 对象具有 person 的所有属性和方法，还有一个新方法叫 sayHi()。

寄生式继承同样适合主要关注对象，而不在乎类型和构造函数的场景。object()函数不是寄生式继承所必需的，任何返回新对象的函数都可以在这里使用。

> **注意**　通过寄生式继承给对象添加函数会导致函数难以重用，与构造函数模式类似。

8.3.6 寄生式组合继承

组合继承其实也存在效率问题。最主要的效率问题就是父类构造函数始终会被调用两次：一次在是创建子类原型时调用，另一次是在子类构造函数中调用。本质上，子类原型最终是要包含超类对象的所有实例属性，子类构造函数只要在执行时重写自己的原型就行了。再来看一看这个组合继承的例子：

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

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

function SubType(name, age){
  SuperType.call(this, name);    // 第二次调用 SuperType()

  this.age = age;
}

SubType.prototype = new SuperType();    // 第一次调用 SuperType()
SubType.prototype.constructor = SubType;
SubType.prototype.sayAge = function() {
  console.log(this.age);
};
```

代码中加粗的部分是调用 SuperType 构造函数的地方。在上面的代码执行后，SubType.prototype 上会有两个属性：name 和 colors。它们都是 SuperType 的实例属性，但现在成为了 SubType 的原型属性。在调用 SubType 构造函数时，也会调用 SuperType 构造函数，这一次会在新对象上创建实例属性 name 和 colors。这两个实例属性会遮蔽原型上同名的属性。图 8-6 展示了这个过程。

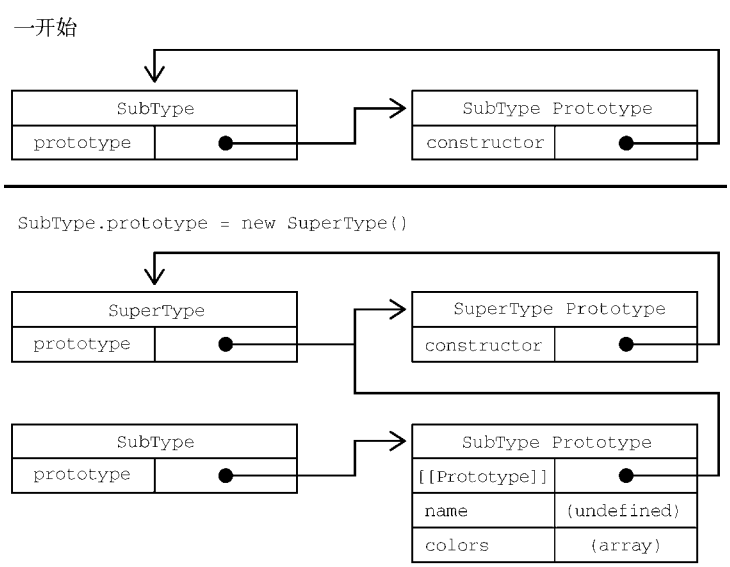

图 8-6

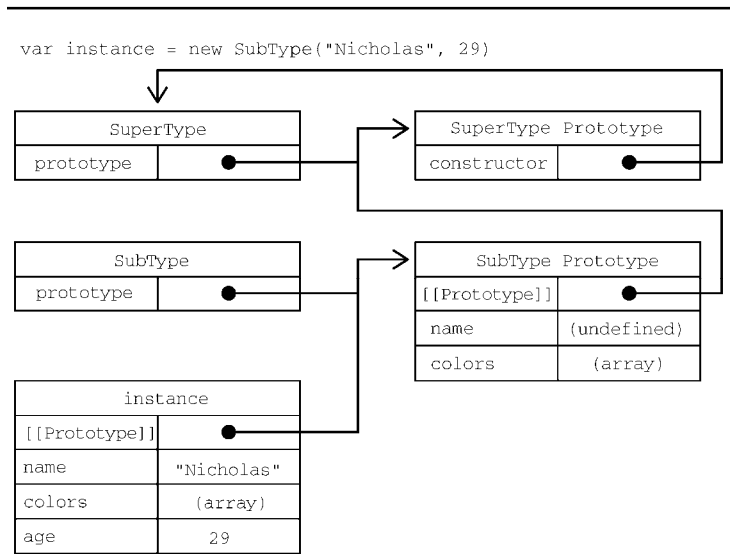

图　8-6（续）

如图 8-6 所示，有两组 name 和 colors 属性：一组在实例上，另一组在 SubType 的原型上。这是调用两次 SuperType 构造函数的结果。好在有办法解决这个问题。

寄生式组合继承通过盗用构造函数继承属性，但使用混合式原型链继承方法。基本思路是不通过调用父类构造函数给子类原型赋值，而是取得父类原型的一个副本。说到底就是使用寄生式继承来继承父类原型，然后将返回的新对象赋值给子类原型。寄生式组合继承的基本模式如下所示：

```
function inheritPrototype(subType, superType) {
  let prototype = object(superType.prototype);  // 创建对象
  prototype.constructor = subType;              // 增强对象
  subType.prototype = prototype;                // 赋值对象
}
```

这个 inheritPrototype()函数实现了寄生式组合继承的核心逻辑。这个函数接收两个参数：子类构造函数和父类构造函数。在这个函数内部，第一步是创建父类原型的一个副本。然后，给返回的 prototype 对象设置 constructor 属性，解决由于重写原型导致默认 constructor 丢失的问题。最后将新创建的对象赋值给子类型的原型。如下例所示，调用 inheritPrototype()就可以实现前面例子中的子类型原型赋值：

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

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

function SubType(name, age) {
  SuperType.call(this, name);
```

```
  this.age = age;
}

inheritPrototype(SubType, SuperType);

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

这里只调用了一次 SuperType 构造函数，避免了 SubType.prototype 上不必要也用不到的属性，因此可以说这个例子的效率更高。而且，原型链仍然保持不变，因此 instanceof 操作符和 isPrototypeOf()方法正常有效。寄生式组合继承可以算是引用类型继承的最佳模式。

8.4 类

前几节深入讲解了如何只使用 ECMAScript 5 的特性来模拟类似于类（class-like）的行为。不难看出，各种策略都有自己的问题，也有相应的妥协。正因为如此，实现继承的代码也显得非常冗长和混乱。

为解决这些问题，ECMAScript 6 新引入的 class 关键字具有正式定义类的能力。类（class）是 ECMAScript 中新的基础性语法糖结构，因此刚开始接触时可能会不太习惯。虽然 ECMAScript 6 类表面上看起来可以支持正式的面向对象编程，但实际上它背后使用的仍然是原型和构造函数的概念。

8.4.1 类定义

与函数类型相似，定义类也有两种主要方式：类声明和类表达式。这两种方式都使用 class 关键字加大括号：

```
// 类声明
class Person {}

// 类表达式
const Animal = class {};
```

与函数表达式类似，类表达式在它们被求值前也不能引用。不过，与函数定义不同的是，虽然函数声明可以提升，但类定义不能：

```
console.log(FunctionExpression);   // undefined
var FunctionExpression = function() {};
console.log(FunctionExpression);   // function() {}

console.log(FunctionDeclaration);  // FunctionDeclaration() {}
function FunctionDeclaration() {}
console.log(FunctionDeclaration);  // FunctionDeclaration() {}

console.log(ClassExpression);      // undefined
var ClassExpression = class {};
console.log(ClassExpression);      // class {}

console.log(ClassDeclaration);     // ReferenceError: ClassDeclaration is not defined
class ClassDeclaration {}
console.log(ClassDeclaration);     // class ClassDeclaration {}
```

另一个跟函数声明不同的地方是，函数受函数作用域限制，而类受块作用域限制：

```
{
  function FunctionDeclaration() {}
  class ClassDeclaration {}
}

console.log(FunctionDeclaration); // FunctionDeclaration() {}
console.log(ClassDeclaration);    // ReferenceError: ClassDeclaration is not defined
```

类的构成

类可以包含构造函数方法、实例方法、获取函数、设置函数和静态类方法，但这些都不是必需的。空的类定义照样有效。默认情况下，类定义中的代码都在严格模式下执行。

与函数构造函数一样，多数编程风格都建议类名的首字母要大写，以区别于通过它创建的实例（比如，通过 `class Foo {}`创建实例 `foo`）：

```
// 空类定义，有效
class Foo {}

// 有构造函数的类，有效
class Bar {
  constructor() {}
}

// 有获取函数的类，有效
class Baz {
  get myBaz() {}
}

// 有静态方法的类，有效
class Qux {
  static myQux() {}
}
```

类表达式的名称是可选的。在把类表达式赋值给变量后，可以通过 `name` 属性取得类表达式的名称字符串。但不能在类表达式作用域外部访问这个标识符。

```
let Person = class PersonName {
  identify() {
    console.log(Person.name, PersonName.name);
  }
}

let p = new Person();

p.identify();               // PersonName PersonName

console.log(Person.name);   // PersonName
console.log(PersonName);    // ReferenceError: PersonName is not defined
```

8.4.2　类构造函数

`constructor` 关键字用于在类定义块内部创建类的构造函数。方法名 `constructor` 会告诉解释器在使用 `new` 操作符创建类的新实例时，应该调用这个函数。构造函数的定义不是必需的，不定义构造函数相当于将构造函数定义为空函数。

1. 实例化

使用 new 操作符实例化 Person 的操作等于使用 new 调用其构造函数。唯一可感知的不同之处就是，JavaScript 解释器知道使用 new 和类意味着应该使用 constructor 函数进行实例化。

使用 new 调用类的构造函数会执行如下操作。

(1) 在内存中创建一个新对象。

(2) 这个新对象内部的[[Prototype]]指针被赋值为构造函数的 prototype 属性。

(3) 构造函数内部的 this 被赋值为这个新对象（即 this 指向新对象）。

(4) 执行构造函数内部的代码（给新对象添加属性）。

(5) 如果构造函数返回非空对象，则返回该对象；否则，返回刚创建的新对象。

来看下面的例子：

```
class Animal {}

class Person {
  constructor() {
    console.log('person ctor');
  }
}

class Vegetable {
  constructor() {
    this.color = 'orange';
  }
}

let a = new Animal();

let p = new Person();  // person ctor

let v = new Vegetable();
console.log(v.color);  // orange
```

类实例化时传入的参数会用作构造函数的参数。如果不需要参数，则类名后面的括号也是可选的：

```
class Person {
  constructor(name) {
    console.log(arguments.length);
    this.name = name || null;
  }
}

let p1 = new Person;          // 0
console.log(p1.name);         // null

let p2 = new Person();        // 0
console.log(p2.name);         // null

let p3 = new Person('Jake');  // 1
console.log(p3.name);         // Jake
```

默认情况下，类构造函数会在执行之后返回 this 对象。构造函数返回的对象会被用作实例化的对象，如果没有什么引用新创建的 this 对象，那么这个对象会被销毁。不过，如果返回的不是 this 对象，而是其他对象，那么这个对象不会通过 instanceof 操作符检测出跟类有关联，因为这个对象的原

型指针并没有被修改。

```
class Person {
  constructor(override) {
    this.foo = 'foo';
    if (override) {
      return {
        bar: 'bar'
      };
    }
  }
}

let p1 = new Person(),
    p2 = new Person(true);

console.log(p1);                      // Person{ foo: 'foo' }
console.log(p1 instanceof Person);  // true

console.log(p2);                      // { bar: 'bar' }
console.log(p2 instanceof Person);  // false
```

类构造函数与构造函数的主要区别是，调用类构造函数必须使用 new 操作符。而普通构造函数如果不使用 new 调用，那么就会以全局的 this（通常是 window）作为内部对象。调用类构造函数时如果忘了使用 new 则会抛出错误：

```
function Person() {}

class Animal {}

// 把 window 作为 this 来构建实例
let p = Person();

let a = Animal();
// TypeError: class constructor Animal cannot be invoked without 'new'
```

类构造函数没有什么特殊之处，实例化之后，它会成为普通的实例方法（但作为类构造函数，仍然要使用 new 调用）。因此，实例化之后可以在实例上引用它：

```
class Person {}

// 使用类创建一个新实例
let p1 = new Person();

p1.constructor();
// TypeError: Class constructor Person cannot be invoked without 'new'

// 使用对类构造函数的引用创建一个新实例
let p2 = new p1.constructor();
```

2. 把类当成特殊函数

ECMAScript 中没有正式的类这个类型。从各方面来看，ECMAScript 类就是一种特殊函数。声明一个类之后，通过 typeof 操作符检测类标识符，表明它是一个函数：

```
class Person {}

console.log(Person);         // class Person {}
console.log(typeof Person);  // function
```

类标识符有 prototype 属性，而这个原型也有一个 constructor 属性指向类自身：

```
class Person{}

console.log(Person.prototype);                          // { constructor: f() }
console.log(Person === Person.prototype.constructor);  // true
```

与普通构造函数一样，可以使用 instanceof 操作符检查构造函数原型是否存在于实例的原型链中：

```
class Person {}

let p = new Person();

console.log(p instanceof Person); // true
```

由此可知，可以使用 instanceof 操作符检查一个对象与类构造函数，以确定这个对象是不是类的实例。只不过此时的类构造函数要使用类标识符，比如，在前面的例子中要检查 p 和 Person。

如前所述，类本身具有与普通构造函数一样的行为。在类的上下文中，类本身在使用 new 调用时就会被当成构造函数。重点在于，类中定义的 constructor 方法**不会**被当成构造函数，在对它使用 instanceof 操作符时会返回 false。但是，如果在创建实例时直接将类构造函数当成普通构造函数来使用，那么 instanceof 操作符的返回值会反转：

```
class Person {}

let p1 = new Person();

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

let p2 = new Person.constructor();

console.log(p2.constructor === Person);         // false
console.log(p2 instanceof Person);              // false
console.log(p2 instanceof Person.constructor);  // true
```

类是 JavaScript 的一等公民，因此可以像其他对象或函数引用一样把类作为参数传递：

```
// 类可以像函数一样在任何地方定义，比如在数组中
let classList = [
  class {
    constructor(id) {
      this.id_ = id;
      console.log(`instance ${this.id_}`);
    }
  }
];

function createInstance(classDefinition, id) {
  return new classDefinition(id);
}

let foo = createInstance(classList[0], 3141);  // instance 3141
```

与立即调用函数表达式相似，类也可以立即实例化：

```
// 因为是一个类表达式，所以类名是可选的
let p = new class Foo {
```

```
  constructor(x) {
    console.log(x);
  }
}('bar');        // bar

console.log(p);  // Foo {}
```

8.4.3　实例、原型和类成员

类的语法可以非常方便地定义应该存在于实例上的成员、应该存在于原型上的成员，以及应该存在于类本身的成员。

1. 实例成员

每次通过new调用类标识符时，都会执行类构造函数。在这个函数内部，可以为新创建的实例（this）添加“自有”属性。至于添加什么样的属性，则没有限制。另外，在构造函数执行完毕后，仍然可以给实例继续添加新成员。

每个实例都对应一个唯一的成员对象，这意味着所有成员都不会在原型上共享：

```
class Person {
  constructor() {
    // 这个例子先使用对象包装类型定义一个字符串
    // 为的是在下面测试两个对象的相等性
    this.name = new String('Jack');

    this.sayName = () => console.log(this.name);

    this.nicknames = ['Jake', 'J-Dog']
  }
}

let p1 = new Person(),
    p2 = new Person();

p1.sayName(); // Jack
p2.sayName(); // Jack

console.log(p1.name === p2.name);            // false
console.log(p1.sayName === p2.sayName);      // false
console.log(p1.nicknames === p2.nicknames);  // false

p1.name = p1.nicknames[0];
p2.name = p2.nicknames[1];

p1.sayName();  // Jake
p2.sayName();  // J-Dog
```

2. 原型方法与访问器

为了在实例间共享方法，类定义语法把在类块中定义的方法作为原型方法。

```
class Person {
  constructor() {
    // 添加到 this 的所有内容都会存在于不同的实例上
    this.locate = () => console.log('instance');
  }
```

```
  // 在类块中定义的所有内容都会定义在类的原型上
  locate() {
    console.log('prototype');
  }
}

let p = new Person();

p.locate();                  // instance
Person.prototype.locate();  // prototype
```

可以把方法定义在类构造函数中或者类块中，但不能在类块中给原型添加原始值或对象作为成员数据：

```
class Person {
  name: 'Jake'
}
// Uncaught SyntaxError: Unexpected token
```

类方法等同于对象属性，因此可以使用字符串、符号或计算的值作为键：

```
const symbolKey = Symbol('symbolKey');

class Person {

  stringKey() {
    console.log('invoked stringKey');
  }
   [symbolKey]() {
    console.log('invoked symbolKey');
  }
   ['computed' + 'Key']() {
    console.log('invoked computedKey');
  }
}

let p = new Person();

p.stringKey();    // invoked stringKey
p[symbolKey]();   // invoked symbolKey
p.computedKey();  // invoked computedKey
```

类定义也支持获取和设置访问器。语法与行为跟普通对象一样：

```
class Person {
  set name(newName) {
    this.name_ = newName;
  }

  get name() {
    return this.name_;
  }
}

let p = new Person();
p.name = 'Jake';
console.log(p.name); // Jake
```

3. 静态类方法

可以在类上定义静态方法。这些方法通常用于执行不特定于实例的操作，也不要求存在类的实例。