与原型成员类似，静态成员每个类上只能有一个。

静态类成员在类定义中使用 static 关键字作为前缀。在静态成员中，this 引用类自身。其他所有约定跟原型成员一样：

```
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', this);
  }

  // 定义在类的原型对象上
  locate() {
    console.log('prototype', this);
  }

  // 定义在类本身上
  static locate() {
    console.log('class', this);
  }
}

let p = new Person();

p.locate();                 // instance, Person {}
Person.prototype.locate();  // prototype, {constructor: ... }
Person.locate();            // class, class Person {}
```

静态类方法非常适合作为实例工厂：

```
class Person {
  constructor(age) {
    this.age_ = age;
  }

  sayAge() {
    console.log(this.age_);
  }

  static create() {
    // 使用随机年龄创建并返回一个 Person 实例
    return new Person(Math.floor(Math.random()*100));
  }
}

console.log(Person.create()); // Person { age_: ... }
```

4. 非函数原型和类成员

虽然类定义并不显式支持在原型或类上添加成员数据，但在类定义外部，可以手动添加：

```
class Person {
  sayName() {
    console.log(`${Person.greeting} ${this.name}`);
  }
}

// 在类上定义数据成员
Person.greeting = 'My name is';
```

```
// 在原型上定义数据成员
Person.prototype.name = 'Jake';

let p = new Person();
p.sayName();  // My name is Jake
```

> **注意** 类定义中之所以没有显式支持添加数据成员，是因为在共享目标（原型和类）上添加可变（可修改）数据成员是一种反模式。一般来说，对象实例应该独自拥有通过 this 引用的数据。

5. 迭代器与生成器方法

类定义语法支持在原型和类本身上定义生成器方法：

```
class Person {
  // 在原型上定义生成器方法
  *createNicknameIterator() {
    yield 'Jack';
    yield 'Jake';
    yield 'J-Dog';
  }

  // 在类上定义生成器方法
  static *createJobIterator() {
    yield 'Butcher';
    yield 'Baker';
    yield 'Candlestick maker';
  }
}

let jobIter = Person.createJobIterator();
console.log(jobIter.next().value);  // Butcher
console.log(jobIter.next().value);  // Baker
console.log(jobIter.next().value);  // Candlestick maker

let p = new Person();
let nicknameIter = p.createNicknameIterator();
console.log(nicknameIter.next().value);  // Jack
console.log(nicknameIter.next().value);  // Jake
console.log(nicknameIter.next().value);  // J-Dog
```

因为支持生成器方法，所以可以通过添加一个默认的迭代器，把类实例变成可迭代对象：

```
class Person {
  constructor() {
    this.nicknames = ['Jack', 'Jake', 'J-Dog'];
  }

  *[Symbol.iterator]() {
    yield *this.nicknames.entries();
  }
}

let p = new Person();
for (let [idx, nickname] of p) {
  console.log(nickname);
}
```

```
// Jack
// Jake
// J-Dog
```

也可以只返回迭代器实例：

```
class Person {
  constructor() {
    this.nicknames = ['Jack', 'Jake', 'J-Dog'];
  }

  [Symbol.iterator]() {
    return this.nicknames.entries();
  }
}

let p = new Person();
for (let [idx, nickname] of p) {
  console.log(nickname);
}
// Jack
// Jake
// J-Dog
```

8.4.4　继承

本章前面花了大量篇幅讨论如何使用 ES5 的机制实现继承。ECMAScript 6 新增特性中最出色的一个就是原生支持了类继承机制。虽然类继承使用的是新语法，但背后依旧使用的是原型链。

1. 继承基础

ES6 类支持单继承。使用 `extends` 关键字，就可以继承任何拥有`[[Construct]]`和原型的对象。很大程度上，这意味着不仅可以继承一个类，也可以继承普通的构造函数（保持向后兼容）：

```
class Vehicle {}

// 继承类
class Bus extends Vehicle {}

let b = new Bus();
console.log(b instanceof Bus);      // true
console.log(b instanceof Vehicle);  // true


function Person() {}

// 继承普通构造函数
class Engineer extends Person {}

let e = new Engineer();
console.log(e instanceof Engineer);  // true
console.log(e instanceof Person);    // true
```

派生类都会通过原型链访问到类和原型上定义的方法。`this` 的值会反映调用相应方法的实例或者类：

```
class Vehicle {
  identifyPrototype(id) {
    console.log(id, this);
  }
```

```
  static identifyClass(id) {
    console.log(id, this);
  }
}

class Bus extends Vehicle {}

let v = new Vehicle();
let b = new Bus();

b.identifyPrototype('bus');       // bus, Bus {}
v.identifyPrototype('vehicle');   // vehicle, Vehicle {}

Bus.identifyClass('bus');         // bus, class Bus {}
Vehicle.identifyClass('vehicle'); // vehicle, class Vehicle {}
```

> **注意** extends 关键字也可以在类表达式中使用，因此 let Bar = class extends Foo {} 是有效的语法。

2. 构造函数、HomeObject 和 super()

派生类的方法可以通过 super 关键字引用它们的原型。这个关键字只能在派生类中使用，而且仅限于类构造函数、实例方法和静态方法内部。在类构造函数中使用 super 可以调用父类构造函数。

```
class Vehicle {
  constructor() {
    this.hasEngine = true;
  }
}

class Bus extends Vehicle {
  constructor() {
    // 不要在调用 super()之前引用 this，否则会抛出 ReferenceError

    super(); // 相当于 super.constructor()

    console.log(this instanceof Vehicle);  // true
    console.log(this);                     // Bus { hasEngine: true }
  }
}

new Bus();
```

在静态方法中可以通过 super 调用继承的类上定义的静态方法：

```
class Vehicle {
  static identify() {
    console.log('vehicle');
  }
}

class Bus extends Vehicle {
  static identify() {
    super.identify();
  }
}

Bus.identify();  // vehicle
```

> **注意**　ES6 给类构造函数和静态方法添加了内部特性[[HomeObject]]，这个特性是一个指针，指向定义该方法的对象。这个指针是自动赋值的，而且只能在 JavaScript 引擎内部访问。super 始终会定义为[[HomeObject]]的原型。

在使用 super 时要注意几个问题。

- super 只能在派生类构造函数和静态方法中使用。

```
class Vehicle {
  constructor() {
    super();
    // SyntaxError: 'super' keyword unexpected
  }
}
```

- 不能单独引用 super 关键字，要么用它调用构造函数，要么用它引用静态方法。

```
class Vehicle {}

class Bus extends Vehicle {
  constructor() {
    console.log(super);
    // SyntaxError: 'super' keyword unexpected here
  }
}
```

- 调用 super()会调用父类构造函数，并将返回的实例赋值给 this。

```
class Vehicle {}

class Bus extends Vehicle {
  constructor() {
    super();

    console.log(this instanceof Vehicle);
  }
}

new Bus(); // true
```

- super()的行为如同调用构造函数，如果需要给父类构造函数传参，则需要手动传入。

```
class Vehicle {
  constructor(licensePlate) {
    this.licensePlate = licensePlate;
  }
}

class Bus extends Vehicle {
  constructor(licensePlate) {
    super(licensePlate);
  }
}

console.log(new Bus('1337H4X')); // Bus { licensePlate: '1337H4X' }
```

- 如果没有定义类构造函数，在实例化派生类时会调用 super()，而且会传入所有传给派生类的参数。

```
class Vehicle {
  constructor(licensePlate) {
    this.licensePlate = licensePlate;
  }
}

class Bus extends Vehicle {}

console.log(new Bus('1337H4X')); // Bus { licensePlate: '1337H4X' }
```

- 在类构造函数中，不能在调用 `super()` 之前引用 `this`。

```
class Vehicle {}

class Bus extends Vehicle {
  constructor() {
    console.log(this);
  }
}

new Bus();
// ReferenceError: Must call super constructor in derived class
// before accessing 'this' or returning from derived constructor
```

- 如果在派生类中显式定义了构造函数，则要么必须在其中调用 `super()`，要么必须在其中返回一个对象。

```
class Vehicle {}

class Car extends Vehicle {}

class Bus extends Vehicle {
  constructor() {
    super();
  }
}

class Van extends Vehicle {
  constructor() {
    return {};
  }
}

console.log(new Car());  // Car {}
console.log(new Bus());  // Bus {}
console.log(new Van());  // {}
```

3. 抽象基类

有时候可能需要定义这样一个类，它可供其他类继承，但本身不会被实例化。虽然 ECMAScript 没有专门支持这种类的语法，但通过 `new.target` 也很容易实现。`new.target` 保存通过 `new` 关键字调用的类或函数。通过在实例化时检测 `new.target` 是不是抽象基类，可以阻止对抽象基类的实例化：

```
// 抽象基类
class Vehicle {
  constructor() {
    console.log(new.target);
    if (new.target === Vehicle) {
      throw new Error('Vehicle cannot be directly instantiated');
```

```
    }
  }
}

// 派生类
class Bus extends Vehicle {}

new Bus();       // class Bus {}
new Vehicle();   // class Vehicle {}
// Error: Vehicle cannot be directly instantiated
```

另外，通过在抽象基类构造函数中进行检查，可以要求派生类必须定义某个方法。因为原型方法在调用类构造函数之前就已经存在了，所以可以通过 this 关键字来检查相应的方法：

```
// 抽象基类
class Vehicle {
  constructor() {
    if (new.target === Vehicle) {
      throw new Error('Vehicle cannot be directly instantiated');
    }

    if (!this.foo) {
      throw new Error('Inheriting class must define foo()');
    }

    console.log('success!');
  }
}

// 派生类
class Bus extends Vehicle {
  foo() {}
}

// 派生类
class Van extends Vehicle {}

new Bus(); // success!
new Van(); // Error: Inheriting class must define foo()
```

4. 继承内置类型

ES6 类为继承内置引用类型提供了顺畅的机制，开发者可以方便地扩展内置类型：

```
class SuperArray extends Array {
  shuffle() {
    // 洗牌算法
    for (let i = this.length - 1; i > 0; i--) {
      const j = Math.floor(Math.random() * (i + 1));
      [this[i], this[j]] = [this[j], this[i]];
    }
  }
}

let a = new SuperArray(1, 2, 3, 4, 5);

console.log(a instanceof Array);       // true
console.log(a instanceof SuperArray);  // true
```

```
console.log(a);  // [1, 2, 3, 4, 5]
a.shuffle();
console.log(a);  // [3, 1, 4, 5, 2]
```

有些内置类型的方法会返回新实例。默认情况下，返回实例的类型与原始实例的类型是一致的：

```
class SuperArray extends Array {}

let a1 = new SuperArray(1, 2, 3, 4, 5);
let a2 = a1.filter(x => !!(x%2))

console.log(a1);  // [1, 2, 3, 4, 5]
console.log(a2);  // [1, 3, 5]
console.log(a1 instanceof SuperArray);  // true
console.log(a2 instanceof SuperArray);  // true
```

如果想覆盖这个默认行为，则可以覆盖 `Symbol.species` 访问器，这个访问器决定在创建返回的实例时使用的类：

```
class SuperArray extends Array {
  static get [Symbol.species]() {
    return Array;
  }
}

let a1 = new SuperArray(1, 2, 3, 4, 5);
let a2 = a1.filter(x => !!(x%2))

console.log(a1);  // [1, 2, 3, 4, 5]
console.log(a2);  // [1, 3, 5]
console.log(a1 instanceof SuperArray);  // true
console.log(a2 instanceof SuperArray);  // false
```


5. 类混入

把不同类的行为集中到一个类是一种常见的 JavaScript 模式。虽然 ES6 没有显式支持多类继承，但通过现有特性可以轻松地模拟这种行为。

> **注意** `Object.assign()`方法是为了混入对象行为而设计的。只有在需要混入类的行为时才有必要自己实现混入表达式。如果只是需要混入多个对象的属性，那么使用`Object.assign()`就可以了。

在下面的代码片段中，`extends` 关键字后面是一个 JavaScript 表达式。任何可以解析为一个类或一个构造函数的表达式都是有效的。这个表达式会在求值类定义时被求值：

```
class Vehicle {}

function getParentClass() {
  console.log('evaluated expression');
  return Vehicle;
}

class Bus extends getParentClass() {}
// 可求值的表达式
```

混入模式可以通过在一个表达式中连缀多个混入元素来实现，这个表达式最终会解析为一个可以被继承的类。如果 `Person` 类需要组合 A、B、C，则需要某种机制实现 B 继承 A，C 继承 B，而 `Person`

再继承 C，从而把 A、B、C 组合到这个超类中。实现这种模式有不同的策略。

一个策略是定义一组“可嵌套”的函数，每个函数分别接收一个超类作为参数，而将混入类定义为这个参数的子类，并返回这个类。这些组合函数可以连缀调用，最终组合成超类表达式：

```
class Vehicle {}

let FooMixin = (Superclass) => class extends Superclass {
  foo() {
    console.log('foo');
  }
};
let BarMixin = (Superclass) => class extends Superclass {
  bar() {
    console.log('bar');
  }
};
let BazMixin = (Superclass) => class extends Superclass {
  baz() {
    console.log('baz');
  }
};

class Bus extends FooMixin(BarMixin(BazMixin(Vehicle))) {}

let b = new Bus();
b.foo();  // foo
b.bar();  // bar
b.baz();  // baz
```

通过写一个辅助函数，可以把嵌套调用展开：

```
class Vehicle {}

let FooMixin = (Superclass) => class extends Superclass {
  foo() {
    console.log('foo');
  }
};
let BarMixin = (Superclass) => class extends Superclass {
  bar() {
    console.log('bar');
  }
};
let BazMixin = (Superclass) => class extends Superclass {
  baz() {
    console.log('baz');
  }
};

function mix(BaseClass, ...Mixins) {
  return Mixins.reduce((accumulator, current) => current(accumulator), BaseClass);
}

class Bus extends mix(Vehicle, FooMixin, BarMixin, BazMixin) {}

let b = new Bus();
b.foo();  // foo
b.bar();  // bar
b.baz();  // baz
```

> **注意** 很多 JavaScript 框架（特别是 React）已经抛弃混入模式，转向了组合模式（把方法提取到独立的类和辅助对象中，然后把它们组合起来，但不使用继承）。这反映了那个众所周知的软件设计原则："组合胜过继承（composition over inheritance）。"这个设计原则被很多人遵循，在代码设计中能提供极大的灵活性。

8.5 小结

对象在代码执行过程中的任何时候都可以被创建和增强，具有极大的动态性，并不是严格定义的实体。下面的模式适用于创建对象。

- 工厂模式就是一个简单的函数，这个函数可以创建对象，为它添加属性和方法，然后返回这个对象。这个模式在构造函数模式出现后就很少用了。
- 使用构造函数模式可以自定义引用类型，可以使用 `new` 关键字像创建内置类型实例一样创建自定义类型的实例。不过，构造函数模式也有不足，主要是其成员无法重用，包括函数。考虑到函数本身是松散的、弱类型的，没有理由让函数不能在多个对象实例间共享。
- 原型模式解决了成员共享的问题，只要是添加到构造函数 `prototype` 上的属性和方法就可以共享。而组合构造函数和原型模式通过构造函数定义实例属性，通过原型定义共享的属性和方法。

JavaScript 的继承主要通过原型链来实现。原型链涉及把构造函数的原型赋值为另一个类型的实例。这样一来，子类就可以访问父类的所有属性和方法，就像基于类的继承那样。原型链的问题是所有继承的属性和方法都会在对象实例间共享，无法做到实例私有。盗用构造函数模式通过在子类构造函数中调用父类构造函数，可以避免这个问题。这样可以让每个实例继承的属性都是私有的，但要求类型只能通过构造函数模式来定义（因为子类不能访问父类原型上的方法）。目前最流行的继承模式是组合继承，即通过原型链继承共享的属性和方法，通过盗用构造函数继承实例属性。

除上述模式之外，还有以下几种继承模式。

- 原型式继承可以无须明确定义构造函数而实现继承，本质上是对给定对象执行浅复制。这种操作的结果之后还可以再进一步增强。
- 与原型式继承紧密相关的是寄生式继承，即先基于一个对象创建一个新对象，然后再增强这个新对象，最后返回新对象。这个模式也被用在组合继承中，用于避免重复调用父类构造函数导致的浪费。
- 寄生组合继承被认为是实现基于类型继承的最有效方式。

ECMAScript 6 新增的类很大程度上是基于既有原型机制的语法糖。类的语法让开发者可以优雅地定义向后兼容的类，既可以继承内置类型，也可以继承自定义类型。类有效地跨越了对象实例、对象原型和对象类之间的鸿沟。