3.4.1 class

尽管 JavaScript 的原型机制并不像传统类那样工作，但这没有阻止要求这个语言扩展其语法糖使其能够更像真正的类那样表达“类”这种强烈的趋势。因此有了 ES6 class 关键字及其相关的机制。

这个特性是具有强烈争议性的讨论结果，代表着关于如何实现 JavaScript 类的几种强烈冲突观点的更小的妥协子集。多数想要 JavaScript 完整类支持的开发者会发现这个新语法的部分十分诱人，同时也会发现还有一些要点仍然缺失。但别着急，TC39 已经开始研究在 ES6 后提供更多的特性来支持类。

新的 ES6 类机制的核心是关键字 class，表示一个**块**，其内容定义了一个函数原型的成员。考虑：

```
class Foo {
    constructor(a,b) {
        this.x = a;
        this.y = b;
    }

    gimmeXY() {
        return this.x * this.y;
    }
}
```

需要注意以下几点。

- class Foo 表明创建一个（具体的）名为 Foo 的函数，与你在前 ES6 中所做的非常类似。
- constructor(..) 指定 Foo(..) 函数的签名以及函数体内容。
- 类方法使用第 2 章讨论过的对象字面量可用的同样的“简洁方法”语法。这也包含本章前面讨论过的简洁生成器形式，以及 ES5 getter/setter 语法。但是，类方法是不可枚举的，而对象方法默认是可枚举的。
- 和对象字面量不一样，在 class 定义体内部不用逗号分隔成员！实际上，这甚至是不允许的。

可以把前面代码中的 class 语法定义粗略理解为下面这个等价前 ES6 代码，之前有过原型风格编码经验的开发者可能会对此非常熟悉：

```
function Foo(a,b) {
    this.x = a;
    this.y = b;
}

Foo.prototype.gimmeXY = function() {
    return this.x * this.y;
}
```

不管是前 ES6 形式还是新的 ES6 `class` 形式，现在都可以按照期望来实例化和使用这个“类”：

```
var f = new Foo( 5, 15 );

f.x;                    // 5
f.y;                    // 15
f.gimmeXY();            // 75
```

注意！尽管 `class Foo` 看起来很像 `function Foo()`，但二者有重要区别。

- 由于前 ES6 可用的 `Foo.call(obj)` 不能工作，`class Foo` 的 `Foo(..)` 调用必须通过 `new` 来实现。
- `function Foo` 是“提升的”（参见本系列《你不知道的 JavaScript（上卷）》第一部分），而 `class Foo` 并不是；`extends ..` 语句指定了一个不能被“提升”的表达式。所以，在实例化一个 `class` 之前必须先声明它。
- 全局作用域中的 `class Foo` 创建了这个作用域的一个词法标识符 `Foo`，但是和 `function Foo` 不一样，并没有创建一个同名的全局对象属性。

因为 `class` 只是创建了一个同名的构造器函数，所以现有的 `instanceof` 运算符对 ES6 类仍然可以工作。然而，ES6 引入了一种使用 `Symbol.hasInstance`（参见 7.3 节）自定义 `instanceof` 如何工作的方法。

还有一种我认为更方便的思考类的方式，就是把它看作一个**宏**（macro），用于自动产生一个 `prototype` 对象。可选的是，如果使用 `extends`，也连接起了 `[[Prototype]]` 关系。

ES6 `class` 本身并不是一个真正的实体，而是一个包裹着其他像函数和属性这样的具体实体并把它们组合到一起的元概念。

除了声明形式，`class` 也可以是一个表达式，就像在这一句中：`var x = class Y { .. }`。对于把类定义（严格说，是构造器本身）作为函数参数传递，或者把它赋给一个对象属性的时候特别有用。

3.4.2 extends 和 super

ES6 类还通过面向类的常用术语 `extends` 提供了一个语法糖，用来在两个函数原型之间建立 `[[Prototype]]` 委托链接——通常被误称为“继承”或者令人迷惑地标识为“原型继承”：

```
class Bar extends Foo {
    constructor(a,b,c) {
        super( a, b );
        this.z = c;
```

```
        }

        gimmeXYZ() {
            return super.gimmeXY() * this.z;
        }
    }

    var b = new Bar( 5, 15, 25 );

    b.x;                        // 5
    b.y;                        // 15
    b.z;                        // 25
    b.gimmeXYZ();               // 1875
```

还有一个重要的新增特性是 `super`，这在前 ES6 中实际上是无法直接支持的（如果不使用某种不幸的 hack 权衡的话）。在构造器中，`super` 自动指向“父构造器”，在前面的例子中就是 `Foo(..)`。在方法中，`super` 会指向“父对象”，这样就可以访问其属性 / 方法了，比如 `super.gimmeXY()`。

`Bar extends Foo` 的意思当然就是把 `Bar.prototype` 的 `[[Prototype]]` 连接到 `Foo.prototype`。所以，在像 `gimmeXYZ()` 这样的方法中，`super` 具体指 `Foo.prototype`，而在 `Bar` 构造器中 `super` 指的是 `Foo`。

`super` 并不仅限于在 `class` 声明中使用。它还可以在对象字面值中使用，用法和我们这里介绍的几乎一样。参见 2.6.5 节获取更多信息。

1. super 恶龙

`super` 的行为根据其所处的位置不同而有所不同，这一点值得注意。公平地说，绝大多数情况下这不是一个问题。但如果你偏离了狭窄的正轨就会出现意外。

可能会存在这种情况，即你想要在构造器中引用 `Foo.prototype`，比如要直接访问它的某个属性或方法。但是，构造器中不能这样使用 `super`；`super.protytype` 不能工作。简单地说 `super(..)` 意味着调用 `new Foo(..)`，但是实际上并不是指向 `Foo` 自身的一个可用引用。

同样地，你可能想要在非构造器方法内部引用 `Foo(..)` 函数。`super.constructor` 指向函数 `Foo(..)`，但是请注意这个函数**只能**通过 `new` 调用。`new super.constructor` 是合法的，不过它在多数情况下没什么用处，因为你无法让这个调用使用或引用当前的 `this` 对象上下文，而这很可能才是你需要的。

还有，看起来似乎 `super` 像 `this` 一样可以被函数上下文驱动，也就是说，它们都能动态绑定。但是，`super` 并不像 `this` 那样是动态的。构造器或函数在声明时在内部建立了 `super` 引用（在 `class` 声明体内），此时 `super` 是静态绑定到这个特定的类层次上的，不能重载（至少在 ES6 中是这样）。

这意味着什么呢？它意味着如果你习惯于调用一个“类”的方法时通过覆盖它的 this 来从另外一个类中“借来”这个方法，比如通过 call() 或 apply(..)，那么如果借用的这个方法内有一个 super，结果很可能出乎意料。考虑下面这个类层次：

```
class ParentA {
    constructor() { this.id = "a"; }
    foo() { console.log( "ParentA:", this.id ); }
}

class ParentB {
    constructor() { this.id = "b"; }
    foo() { console.log( "ParentB:", this.id ); }
}

class ChildA extends ParentA {
    foo() {
        super.foo();
        console.log( "ChildA:", this.id );
    }
}

class ChildB extends ParentB {
    foo() {
        super.foo();
        console.log( "ChildB:", this.id );
    }
}

var a = new ChildA();
a.foo();                // ParentA: a
                        // ChildA: a
var b = new ChildB();   // ParentB: b
b.foo();                // ChildB: b
```

在前面的代码中，看起来一切似乎都自然而然。但如果你想要借用 b.foo() 并在 a 的上下文中使用它——通过动态 this 绑定，这样的借用是十分常见的，并且有多种不同的使用方法，包括最著名 mixins——你可能会发现结果不幸地出乎意料：

```
// 在a的上下文中借来b.foo()使用
b.foo.call( a );            // ParentB: a
                            // ChildB: a
```

可以看到，this.id 引用被动态重新绑定，因此两种情况下都打印： a，而不是： b。但是 b.foo() 的 super.foo() 引用没有被动态重绑定，所以它仍然打印出 ParentB 而不是期望的 ParentA。

因为 b.foo() 引用了 super，它是静态绑定到 ChildB/ParentB 类层次的，而不能用在 ChildA / ParentA 类层次。ES6 并没有对这个局限提供解决方案。

如果你有一个静态类层次，而没有“异花传粉”的话，super 似乎可以按照直觉期望

工作。但公正地说，实现 this 感知编码的主要好处之一就是这种灵活性。简单地说，class+super 要求避免使用这样的技术。

最后的选择归结为把对象设计限制在这些静态类层次之内——class、extends 和 super 就好——或者放弃“模拟”类的企图转而拥抱动态和灵活性，拥抱非类对象和 [[Prototype]] 委托（参见本系列《你不知道的 JavaScript（上卷）》第二部分）。

2. 子类构造器

对于类和子类来说，构造器并不是必须的；如果省略的话那么二者都会自动提供一个默认构造器。但是，这个默认替代构造器对于直接类和扩展类来说有所不同。

具体来说，默认子类构造器自动调用父类的构造器并传递所有参数。换句话说，可以把默认子类构造器看成下面这样：

```
constructor(...args) {
    super(...args);
}
```

这是一个需要注意的重要细节。并不是在所有支持类的语言中子类构造器都会自动调用父类构造器。C++ 会这样，Java 则不然。更重要的是，在前 ES6 的类中，并不存在这样的自动“父类构造器”调用。如果你的代码依赖的这样的调用**没有**发生，那么转换为 ES6 class 的时候要格外小心。

另外一个 ES6 子类构造器或许是出乎意料的偏离 / 限制是：子类构造器中调用 super(..) 之后才能访问 this。其原因比较微妙复杂，但可以归结为创建 / 初始化你的实例 this 的实际上是父构造器。前 ES6 中，它的实现正相反；this 对象是由“子类构造器”创建的，然后在子类的 this 上下文中调用“父类”构造器。

下面来说明一下。以下代码在前 ES6 中可以工作：

```
function Foo() {
    this.a = 1;
}

function Bar() {
    this.b = 2;
    Foo.call( this );
}

// `Bar` "extends" `Foo`
Bar.prototype = Object.create( Foo.prototype );
```

而这个等价 ES6 代码则不合法：

```
class Foo {
    constructor() { this.a = 1; }
```

```
}

class Bar extends Foo {
    constructor() {
        this.b = 2;     // 不允许在super()之前
        super();        // 要改正的话可以交换这两条语句
    }
}
```

对这个例子的修正很简单，只要交换子类 `Bar` 构造器中两个语句的顺序即可。但是，如果你依赖于前 ES6 支持跳过调用“父构造器”这一特性，那么要小心，因为在 ES6 中已经不允许这么做了。

3. 扩展原生类

新的 `class` 和 `extend` 设计带来的最大好处之一是（终于！）可以构建内置类的子类了。比如 `Array`。考虑：

```
class MyCoolArray extends Array {
    first() { return this[0]; }
    last() { return this[this.length - 1]; }
}

var a = new MyCoolArray( 1, 2, 3 );

a.length;                   // 3
a;                          // [1,2,3]

a.first();                  // 1
a.last();                   // 3
```

在 ES6 之前，有一个 `Array` 的伪“子类”通过手动创建对象并链接到 `Array.prototype`，只能部分工作。它不支持真正 array 的特有性质，比如自动更新 `length` 属性。ES6 子类则可以完全按照期望“继承”并新增特性！

另外一个常见的前 ES6“子类”局限是在创建自定义 error“子类”时与 `Error` 对象所相关的。真正的 `Error` 对象创建时，会自动捕获特殊的 `stack` 信息，包括生成错误时的行号和文件名。前 ES6 自定义 error“子类”没有这样的特性，这严重限制了它们的应用。

ES6 前来解救：

```
class Oops extends Error {
    constructor(reason) {
        this.oops = reason;
    }
}

// 之后：
var ouch = new Oops( "I messed up!" );
throw ouch;
```

前面代码中的自定义 error 对象 ouch 的行为就像其他任何真正 error 对象一样，包括捕获 stack。这是一个巨大的改进！

3.4.3 new.target

ES6 以 new.target 的形式引入了一个新概念，称为**元属性**（meta property，参见第 7 章）。

如果说看起来有点奇怪的话，那么实际上也是这样；把关键字通过 . 与属性名关联起来可绝对不是常见的 JavaScript 模式。

new.target 是一个新的在所有函数中都可用的“魔法”值，尽管在一般函数中它通常是 undefined。在任何构造器中，new.target 总是指向 new 实际上直接调用的构造器，即使构造器是在父类中且通过子类构造器用 super(..) 委托调用。

考虑：

```
class Foo {
    constructor() {
        console.log( "Foo: ", new.target.name );
    }
}

class Bar extends Foo {
    constructor() {
        super();
        console.log( "Bar: ", new.target.name );
    }
    baz() {
        console.log( "baz: ", new.target );
    }
}

var a = new Foo();
// Foo: Foo

var b = new Bar();
// Foo: Bar   <-- 遵循new调用点
// Bar: Bar

b.baz();
// baz: undefined
```

除了访问静态属性 / 方法（参见下一小节）之外，类构造器中的 new.target 元属性没有什么其他用处。

如果 new.target 是 undefined，那么你就可以知道这个函数不是通过 new 调用的。因此如果需要的话可以强制一个 new 调用。

3.4.4 static

当子类 Bar 从父类 Foo 扩展的时候，我们已经看到 Bar.prototype 是 [[Prototype]] 链接到 Foo.prototype 的。然而，Bar() 也 [[Prototype]] 链接到 Foo()。这一点可能并不显而易见。

但是，在为一个类声明了 static 方法（不只是属性）的情况下这就很有用了，因为这些是直接添加到这个类的函数对象上的，而不是在这个函数对象的 prototype 对象上。考虑：

```
class Foo {
    static cool() { console.log( "cool" ); }
    wow() { console.log( "wow" ); }
}

class Bar extends Foo {
    static awesome() {
        super.cool();
        console.log( "awesome" );
    }
    neat() {
        super.wow();
        console.log( "neat" );
    }
}

Foo.cool();                 // "cool"

Bar.cool();                 // "cool"
Bar.awesome();              // "cool"
                            // "awesome"
var b = new Bar();
b.neat();                   // "wow"
                            // "neat"

b.awesome;                  // undefined
b.cool;                     // undefined
```

小心不要误以为 static 成员在类的原型链上。实际上它们在函数构造器之间的双向 / 并行链上。

Symbol.species Getter 构造器

static 适用的一个地方就是为派生（子）类设定 Symbol.species getter（规范内称为 @@species）。如果当任何父类方法需要构造一个新实例，但不想使用子类的构造器本身时，这个功能使得子类可以通知父类应该使用哪个构造器。

举例来说，Array 有很多方法会创造并返回一个新的 Array 实例。如果定义一个 Array 的子类，但是想要这些方法仍然构造真正的 Array 实例而不是你的子类实例，就可以这样使用：

```
class MyCoolArray extends Array {
    // 强制species为父构造器
    static get [Symbol.species]() { return Array; }
```

```
}

var a = new MyCoolArray( 1, 2, 3 ),
    b = a.map( function(v){ return v * 2; } );

b instanceof MyCoolArray;   // false
b instanceof Array;         // true
```

下面说明父类方法如何使用子类 species 声明，这有点类似于 `Array#map(..)` 所做的，考虑：

```
class Foo {
    // 推迟species为子构造器
    static get [Symbol.species]() { return this; }
    spawn() {
        return new this.constructor[Symbol.species]();
    }
}

class Bar extends Foo {
    // 强制species为父构造器
    static get [Symbol.species]() { return Foo; }

}
var a = new Foo();
var b = a.spawn();
b instanceof Foo;                   // true

var x = new Bar();
var y = x.spawn();
y instanceof Bar;                   // false
y instanceof Foo;                   // true
```

父类 `Symbol.species` 通过 `return this` 来延迟到子类，就像通常期望的那样。然后 `Bar` 覆盖手动声明使用 `Foo` 来进行实例创建。当然，子类仍然可以使用 `new this.constructor(..)` 来创建自身的实例。

3.5 小结

在代码组织方面，ES6 引入了几个新特性。

- 迭代器提供了对数组或运算的顺序访问。可以通过像 `for..of` 和 `...` 这些新语言特性来消耗迭代器。
- 生成器是支持本地暂停 / 恢复的函数，通过迭代器来控制。它们可以用于编程（也是交互地，通过 `yield/next(..)` 消息传递）**生成**供迭代消耗的值。
- 模块支持对实现细节的私有封装，并提供公开导出 API。模块定义是基于文件的单例实例，在编译时静态决议。
- 类对基于原型的编码提供了更清晰的语法。新增的 `super` 也解决了 `[[Prototype]]` 链中相对引用的棘手问题。

如果想要通过拥抱 ES6 来改进你的 JavaScript 项目架构，那么应该首先考虑这些新工具。

第 4 章

异步流控制

如果你已经编写过大量 JavaScript 代码，那么一定了解异步编程是必需的技能。管理异步的主要机制一直以来都是函数回调。

然而，ES6 增加了一个新的特性来帮助解决只用回调实现异步的严重缺陷：Promise。另外，可以回顾一下（前面章节中的）生成器，研究一种组合使用这两者的模式，这是 JavaScript 异步流控制技术的一大进步。

4.1 Promise

让我们先来理清一些错误观念：Promise 不是对回调的替代。Promise 在回调代码和将要执行这个任务的异步代码之间提供了一种可靠的中间机制来管理回调。

另外一种看待 Promise 的角度是把它看作事件监听者。可以在其上注册以监听某个事件，在任务完成之后得到通知。这是一个只触发一次的事件，但仍然可以被看作事件。

可以把 Promise 链接到一起，这就把一系列异步完成的步骤串联了起来。通过与像 `all(..)` 方法（经典术语中称为“门”）和 `race(..)` 方法（经典术语中称为“latch”）这样更高级的抽象概念结合起来，Promise 链提供了一个近似的异步控制流。

还有一种定义 Promise 的方式，就是把它看作一个**未来值**（future value），对一个值的独立于时间的封装容器。不管这个容器底层的值是否已经最终确定，都可以用同样的方法应用其值。一旦观察到 Promise 的决议就立刻提取出这个值。换句话说，Promise 可以被看作是同步函数返回值的异步版本。