在前面的代码中，通过 Object.create(..) 语句 obj2 [[Prototype]] 链接到了 obj1。而为了创建反向（环）的链接，我们在 obj1 符号位置 Symbol.for("[[Prototype]]")（参见 2.13 节）处创建了属性。这个符号可能看起来有点特殊 / 神奇，但实际上并非如此。它只是给我提供了一个方便的与我正在执行的任务关联的命名钩子，以便语义上引用。

然后，代理的 get(..) 处理函数首先查看这个代理上是否有请求的 key。如果没有，就手动把这个运算转发给保存在 target 的 Symbol.for("[[Prototype]]") 位置中的对象引用。

这种模式的一个重要优点是，obj1 和 obj2 的定义几乎不会受到在它们之间建立的这种环状关系的影响。尽管为了简洁的缘故，前面代码把所有的代码都纠缠到了一起，但是仔细观察可以看到，代理处理函数的逻辑完全是通用的（并不具体了解 obj1 和 obj2 的细节）。所以，这段逻辑可提取出来封装为一个单独的辅助函数，比如 setCircularPrototypeOf(..)。我们把这个实现留给读者作为练习。

既然已经了解了如何通过 get(..) 来模拟一个 [[Prototype]] 链接，现在让我们来深入 hack 一下。不用环状 [[Prototype]]，用多个 [[Prototype]] 链接（也就是“多继承”）怎么样？实际上这非常简单直接：

```
var obj1 = {
        name: "obj-1",
        foo() {
            console.log( "obj1.foo:", this.name );
        },
    },
    obj2 = {
        name: "obj-2",
        foo() {
            console.log( "obj2.foo:", this.name );
        },
        bar() {
            console.log( "obj2.bar:", this.name );
        }
    },
    handlers = {
        get(target,key,context) {
            if (Reflect.has( target, key )) {
                return Reflect.get(
                    target, key, context
                );
            }
            // 伪装多个[[Prototype]]
            else {
                for (var P of target[
                    Symbol.for( "[[Prototype]]" )
                ]) {
                    if (Reflect.has( P, key )) {
                        return Reflect.get(
                            P, key, context
                        );
```

```
                    }
                }
            }
        }
    },
    obj3 = new Proxy(
        {
            name: "obj-3",
            baz() {
                this.foo();
                this.bar();
            }
        },
        handlers
    );

// 伪装多个[[Prototype]]链接
obj3[ Symbol.for( "[[Prototype]]" ) ] = [
    obj1, obj2
];

obj3.baz();
// obj1.foo: obj-3
// obj2.bar: obj-3
```

正如前面环状 [[Prototype]] 例子之后的注释中提到的一样，我们没有实现 set(..) 处理函数，但是要实现一个完整解决方案，模拟 [[Set]] 动作作为普通的 [[Prototype]] 行为是必要的。

obj3 建立了多委托到 obj1 和 obj2。在 obj3.baz() 中，this.foo() 调用最后从 obj1 中提出 foo()（先到先得，虽然 obj2 上也有一个 foo()）。如果我们把链接重新排序为 obj2、obj1，就会找到并使用 obj2.foo()。

而现在 this.bar() 调用不会在 obj1 上找到 bar()，所以它会陷入检查 obj2，在其中找到匹配。

obj1 和 obj2 表示 obj3 的两条平行的 [[Prototype]] 链。obj1 和 / 或 obj2 本身也可以有普通的 [[Prototype]] 委托到其他对象，或者本身也可以是一个多委托的代理（就像 obj3 一样）。

就像前面的环状 [[Prototype]] 链例子一样，obj1、obj2 和 obj3 的定义与通用的处理多委托代理的逻辑几乎是完全分离的。要定义一个像 setPrototypesOf(..)（注意这个表示复数的“s”）这样的工具接收一个主对象和一个对象列表来模拟多 [[Prototype]] 链接是很简单的。我们还是把这个实现留给读者作为练习。

希望在各种各样的例子之后代理的威力现在变得明朗了。代理使得很多其他威力强大的元编程任务成为可能。

7.5 Reflect API

Reflect 对象是一个平凡对象（就像 Math），不像其他内置原生值一样是函数 / 构造器。

它持有对应于各种可控的元编程任务的静态函数。这些函数一对一对应着代理可以定义的处理函数方法（trap）。

这些函数中的一部分看起来和 Object 上的同名函数类似：

- Reflect.getOwnPropertyDescriptor(..);
- Reflect.defineProperty(..);
- Reflect.getPrototypeOf(..);
- Reflect.setPrototypeOf(..);
- Reflect.preventExtensions(..);
- Reflect.isExtensible(..)。

一般来说这些工具和 Object.* 的对应工具行为方式类似。但是，有一个区别是如果第一个参数（目标对象）不是对象的话，Object.* 相应工具会试图把它类型转换为一个对象。而这种情况下 Reflect.* 方法只会抛出一个错误。

可以使用下面这些工具访问 / 查看一个对象键：

Reflect.ownKeys(..)
: 返回所有“拥有”的（不是“继承”的）键的列表，就像 Object.getOwnPropertyNames(..) 和 Object.getOwnPropertySymbols(..) 返回的一样。关于键的顺序参见后面的“属性排序”一节。

Reflect.enumerate(..)
: 返回一个产生所有（拥有的和“继承的”）**可枚举的**（enumerable）非符号键集合的迭代器（参见本系列《你不知道的 JavaScript（上卷）》第二部分）。本质上说，这个键的集合和 foo..in 循环处理的那个键的集合是一样的。关于键的顺序参见后面的“属性排序”一节。

Reflect.has(..)
: 实质上和 in 运算符一样，用于检查某个属性是否在某个对象上或者在它的 [[Prototype]] 链上。比如，Reflect.has(o, "foo") 实质上就是执行 "foo" in o。

函数调用和构造器调用可以通过使用下面这些工具手动执行，与普通的语法（比如，(..) 和 new）分开：

`Reflect.apply(..)`

举例来说，`Reflect.apply(foo,thisObj,[42,"bar"])` 以 `thisObj` 作为 `this` 调用 `foo(..)` 函数，传入参数 `42` 和 `"bar"`。

`Reflect.construct(..)`

举例来说，`Reflect.construct(foo,[42,"bar"])` 实质上就是调用 `new foo(42,"bar")`。

可以使用下面这些工具来手动执行对象属性访问、设置和删除。

`Reflect.get(..)`

举例来说，`Reflect.get(o,"foo")` 提取 `o.foo`。

`Reflect.set(..)`

举例来说，`Reflect.set(o,"foo",42)` 实质上就是执行 `o.foo = 42`。

`Reflect.deleteProperty(..)`

举例来说，`Reflect.deleteProperty(o,"foo")` 实质上就是执行 `delete o.foo`。

`Reflect` 的元编程能力提供了模拟各种语法特性的编程等价物，把之前隐藏的抽象操作暴露出来。比如，你可以利用这些能力扩展功能和 API，以实现**领域特定语言**（DSL）。

属性排序

在 ES6 之前，一个对象键 / 属性的列出顺序是依赖于具体实现，并未在规范中定义。一般来说，多数引擎按照创建的顺序进行枚举，虽然开发者们一直被强烈建议不要依赖于这个顺序。

对于 ES6 来说，拥有属性的列出顺序是由 `[[OwnPropertyKeys]]` 算法定义的（ES6 规范，9.1.12 节），这个算法产生所有拥有的属性（字符串或符号），不管是否可枚举。这个顺序只对 `Reflect.ownKeys(..)`（以及扩展的 `Object.getOwnPropertyNames(..)` 和 `Object.getOwnPropertySymbols(..)`）有保证。

其顺序为：

(1) 首先，按照数字上升排序，枚举所有整数索引拥有的属性；
(2) 然后，按照创建顺序枚举其余的拥有的字符串属性名；
(3) 最后，按照创建顺序枚举拥有的符号属性。

考虑：

```
var o = {};

o[Symbol("c")] = "yay";
```

```
o[2] = true;
o[1] = true;
o.b = "awesome";
o.a = "cool";

Reflect.ownKeys( o );               // [1,2,"b","a",Symbol(c)]
Object.getOwnPropertyNames( o );    // [1,2,"b","a"]
Object.getOwnPropertySymbols( o );  // [Symbol(c)]
```

另一方面，[[Enumerate]] 算法（ES6 规范，9.1.11 节）只从目标对象和它的 [[Prototype]] 链产生可枚举属性。它用于 Reflect.enumerate(..) 和 for..in。可以观察到的顺序和具体的实现相关，不由规范控制。

与之对比，Object.keys(..) 调用 [[OwnPropertyKeys]] 算法取得拥有的所有键的列表。但是，它会过滤掉不可枚举属性，然后把这个列表重新排序来遵循遗留的与实现相关的行为特性，特别是 JSON.stringify(..) 和 for..in。因此通过扩展，这个顺序也和 Reflect.enumerate(..) 顺序相匹配。

换句话说，所有这 4 种机制（Reflect.enumerate(..)、Object.keys(..)、for..in 和 JSON.stringify(..)）都会匹配同样的与具体实现相关的排序，尽管严格上说是通过不同的路径。

把这 4 种机制与 [[OwnPropertyKeys]] 的排序匹配的具体实现是允许的，但并不是必须的。尽管如此，你很可能会看到它们的排序特性是这样的：

```
var o = { a: 1, b: 2 };
var p = Object.create( o );
p.c = 3;
p.d = 4;

for (var prop of Reflect.enumerate( p )) {
    console.log( prop );
}
// c d a b

for (var prop in p) {
    console.log( prop );
}
// c d a b

JSON.stringify( p );
// {"c":3,"d":4}

Object.keys( p );
// ["c","d"]
```

总结一下：对于 ES6 来说，Reflect.ownKeys(..)、Object.getOwnPropertyNames(..) 和 Object.getOwnPropertySymbols(..) 的顺序都是可预测且可靠的，这由规范保证。所以依

赖于这个顺序的代码是安全的。

`Reflect.enumerate(..)`、`Object.keys(..)` 和 `for..in`（以及扩展的 `JSON.stringification(..)`）还像过去一样，可观察的顺序是相同的。但是这个顺序不再必须与 `Reflect.ownKeys(..)` 相同。在使用它们依赖于具体实现的顺序时仍然要小心。

7.6 特性测试

什么是特性测试？就是一种由你运行的用来判断一个特性是否可用的测试。有时候，这个测试不只是为了测试特性是否存在，还是为了测试特性是否符合指定的行为规范——特性可能存在但却是有问题的。

测试程序的运行环境，然后确定程序行为方式，这是一种元编程技术。

JavaScript 中最常用的特性测试是检查一个 API 是否存在，如果不存在的话，定义一个 polyfill（参见第 1 章）。比如：

```
if (!Number.isNaN) {
    Number.isNaN = function(x) {
        return x !== x;
    };
}
```

这段代码中的 `if` 语句是元编程：我们检查程序和它的运行环境，以此来确定是否应该继续以及如何继续。

但是如何测试涉及新语法的特性呢？

可能你会想到使用像下面这样的代码：

```
try {
    a = () => {};
    ARROW_FUNCS_ENABLED = true;
}
catch (err) {
    ARROW_FUNCS_ENABLED = false;
}
```

不幸的是，这行不通，因为我们的 JavaScript 程序是需要编译的。所以，如果引擎还不支持 ES6 箭头函数的话，就会停在 `() => {}` 语法处。这样你程序中的一个语法错误会使其无法运行，你的程序也就无法根据特性是否被支持而作出不同的反应。

要针对语法相关的特性进行特性测试的元编程，我们需要一种方法能够把测试与程序的初始编译步骤隔绝开来。比如，如果我们可以把测试代码放在一个字符串里，那么 JavaScript 引擎默认就不会试图编译这个字符串的内容，直到要求它这么做。

我们直接跳到使用 `eval(..)` 怎么样？

还没那么快，参见本系列《你不知道的 JavaScript（上卷）》第一部分，其中介绍了为什么 `eval(..)` 不是一个好主意。但是还有一种缺点少一些的选择：`Function(..)` 构造器。

考虑：

```
try {
    new Function( "( () => {} )" );
    ARROW_FUNCS_ENABLED = true;
}
catch (err) {
    ARROW_FUNCS_ENABLED = false;
}
```

好吧，现在我们就是在通过元编程确定像箭头函数这样的特性是否能在当前引擎上编译。你可能会想知道，拿到了这个信息又能做什么呢？

有了 API 的存在性检查，并且定义了用作退路的 API polyfill，那么测试成功和失败后的路径是很清晰的。但是知道了 `ARROW_FUNCS_ENABLED` 为 `true` 还是 `false` 这个信息之后又能做什么呢？

引擎不支持的语法特性不能出现在一个文件中，因此，不能在这个文件中分别使用或是不使用这个语法定义不同的函数。

你能做的是，使用这个测试来确定应该加载一组 JavaScript 文件中的哪一个。举例来说，如果在你的 JavaScript 应用程序的引导程序（bootstrapper）中有一组这样的特性测试，就可以通过测试环境来确定你的 ES6 代码是能够直接加载运行，还是需要加载代码的 transpile 版本（参见第 1 章）。

这种技术叫作**分批发布**（split delivery）。

事实表明，有时候你的 ES6 JavaScript 程序能够完整“原生”运行在 ES6+ 浏览器中，但有时候又需要 transpilation 运行在前 ES6 浏览器中。如果总是加载使用 transpile 的代码，甚至在新的 ES6 兼容环境中也是这样，那么至少有时候是在运行非优化的代码。这并不理想。

分批发布更复杂也更高级，但它是一种更成熟、更健壮的方法，可以弥合代码编写和程序运行浏览器支持的特性之间的裂隙。

FeatureTests.io

为所有 ES6+ 语法和语义行为特性定义特性测试，可能是你并不想亲自动手的令人望而却步的工作。因为这些测试需要动态编译（`new Function(..)`），所以会有一些不幸的性能损失。

另外，每次程序运行的时候都要运行这些测试可能也是一种浪费，因为一般用户浏览器最多也只是几个星期才更新一次，即便如此，也不是每次更新都会出现新特性。

最后，管理应用到具体代码的特性测试列表也是容易失控和出错的——你的程序几乎不会使用所有 ES6 特性。

FeatureTests.io（https://featuretests.io）为此提供了解决方案。

你可以在自己的页面中加载这个服务库，然后它会加载最新的测试定义并运行所有特性测试。如果可能的话，就使用 Web Worker 的后台进程实现这些，以减少性能损失。它还会使用 LocalStorage 持久存储缓存结果，并且其实现方式使得你访问的所有使用这个服务的站点都可以共享缓存，这显著降低了每个浏览器实例上运行这些测试的所需频率。

这样你得到了每个用户浏览器的运行时特性测试，然后就可以巧妙使用这些测试结果根据用户的环境为用户提供最合适的代码（不多也不少）。

另外，这个服务提供了工具和 API 来扫描你的文件以确定需要哪些特性，所以你可以完全自动化分批发布构建过程。

FeatureTests.io 使得应用所有 ES6 及之后的特性测试，确保在给定的环境中只加载运行最优代码成为可能。

7.7 尾递归调用（Tail Call Optimization,TCO）

通常，在一个函数内部调用另一个函数的时候，会分配第二个**栈帧**来独立管理第二个函数调用的变量 / 状态。这个分配不但消耗处理时间，也消耗了额外的内存。

通常调用栈链最多有 10~15 个从一个函数到另一个函数的跳转。这种情况下，内存使用并不会造成任何实际问题。

但是，当考虑到递归编程的时候（一个函数重复调用自身）——或者两个或多个函数彼此调用形成递归——调用栈的深度很容易达到成百上千，甚至更多。如果内存的使用无限制地增长下去，你可能看到了它将导致的问题。

JavaScript 引擎不得不设置一个武断的限制来防止这种编程技术引起浏览器和设备内存耗尽而崩溃。这也是为什么达到这个限制的时候我们会得到烦人的“RangeError: Maximum call stack size exceeded”。

调用栈深度限制不由规范控制。它是依赖于具体实现的，并且根据浏览器和设备不同而有所不同。在编码的时候不要对观察到的具体限制值有任何强假定，因为它很可能根据发布版本的不同而有所不同。

有一些称为**尾调用**（tail call）的函数调用模式，可以以避免额外栈帧分配的方式进行优化。如果可以避免额外的分配，就没有理由任意限制调用栈深度，所以引擎就可以不设置这个限制。

尾调用是一个 `return` 函数调用的语句，除了调用后返回其返回值之外没有任何其他动作。

这个优化只在 `strict` 模式下应用。这又是一个要坚持编写 `strict` 模式代码的原因！

下面是一个不在尾位置的函数调用：

```
"use strict";

function foo(x) {
    return x * 2;
}

function bar(x) {
    // 这不是尾调用
    return 1 + foo( x );
}

bar( 10 );              // 21
```

`foo(x)` 调用完毕后还得执行 `1 + ..`，所以 `bar(..)` 调用的状态需要被保留。

但下面代码展示的对 `foo(..)` 和 `bar(..)` 的调用都处于尾位置，因为它们是在其代码路径上发生的最后一件事（除了 `return`）：

```
"use strict";

function foo(x) {
    return x * 2;
}

function bar(x) {
    x = x + 1;
    if (x > 10) {
        return foo( x );
    }
    else {
        return bar( x + 1 );
    }
}

bar( 5 );               // 24
bar( 15 );              // 32
```

在这个程序中，`bar(..)` 显然是递归，而 `foo(..)` 只是一个普通函数调用。在这两种情况下，函数调用都处于**合适的尾位置**（proper tail position）。`x + 1` 在 `bar(..)` 调用之前求值，在调用结束后，所做的只有 `return`。

这些形式的正确尾调用（Proper Tail Call，PTC）是可以被优化的——称为尾调用优化（Tail Call Optimization，TCO）——于是额外的栈帧分配是不需要的。引擎不需要对下一个函数调用创建一个新的栈帧，只需复用已有的栈帧。这能够工作是因为一个函数不需要保留任何当前状态——在 PTC 之后不需要这个状态做任何事情。

TCO 意味着对调用栈的允许深度没有任何限度。对于一般程序中的普通函数调用，这个技巧有些许优化，但更重要的是打开了在程序表达中使用递归的大门，甚至是调用栈的调用深度可能达到成千上万的时候。

现在我们不再只把递归作为解决问题的理论方案了，而是可以实际将其用在 JavaScript 程序中！

对于 ES6 来说，不管是否为递归，所有的 PTC 都应该以这种方式优化。

7.7.1 尾调用重写

但这里的问题是只有 PTC 可以被优化；非 PTC 当然仍然可以工作，但会像以前一样触发栈帧分配。如果你希望这个优化介入的话，需要认真设计函数结构支持 PTC。

如果有一个函数不是以 PTC 方式编写的，那么你可能会需要手动重新安排代码以适合 TCO。

考虑：

```
"use strict";

function foo(x) {
    if (x <= 1) return 1;
    return (x / 2) + foo( x - 1 );
}

foo( 123456 );          // RangeError
```

调用 `foo(x-1)` 不是 PTC，因为它的结果每次在 `return` 之前要加上 `(x / 2)`。

但是，要想使这段代码适合 ES6 引擎 TCO，可以这样重写：

```
"use strict";

var foo = (function(){
    function _foo(acc,x) {
        if (x <= 1) return acc;
        return _foo( (x / 2) + acc, x - 1 );
    }

    return function(x) {
        return _foo( 1, x );
```