5.5.2 未来的模块机制

ES6 中为模块增加了一级语法支持。但通过模块系统进行加载时，ES6 会将文件当作独立的模块来处理。每个模块都可以导入其他模块或特定的 API 成员，同样也可以导出自己的 API 成员。

基于函数的模块并不是一个能被稳定识别的模式（编译器无法识别），它们的 API 语义只有在运行时才会被考虑进来。因此可以在运行时修改一个模块的 API（参考前面关于公共 API 的讨论）。

相比之下，ES6 模块 API 更加稳定（API 不会在运行时改变）。由于编辑器知道这一点，因此可以在（的确也这样做了）编译期检查对导入模块的 API 成员的引用是否*真实存在*。如果 API 引用并不存在，编译器会在运行时抛出一个或多个“早期”错误，而不会像往常一样在运行期采用动态的解决方案。

ES6 的模块没有“行内”格式，必须被定义在独立的文件中（一个文件一个模块）。浏览器或引擎有一个默认的“模块加载器”（可以被重载，但这远超出了我们的讨论范围）可以在导入模块时异步地加载模块文件。

考虑以下代码：

bar.js

```
function hello(who) {
    return "Let me introduce: " + who;
}

export hello;
```

foo.js

```
// 仅从 "bar" 模块导入 hello()
import hello from "bar";

var hungry = "hippo";

function awesome() {
    console.log(
        hello( hungry ).toUpperCase()
    );
}

export awesome;
```

baz.js

```
// 导入完整的 "foo" 和 "bar" 模块
```

```
module foo from "foo";
module bar from "bar";

console.log(
    bar.hello( "rhino" )
); // Let me introduce: rhino

foo.awesome(); // LET ME INTRODUCE: HIPPO
```

需要用前面两个代码片段中的内容分别创建文件 foo.js 和 bar.js。然后如第三个代码片段中展示的那样，bar.js 中的程序会加载或导入这两个模块并使用它们。

`import` 可以将一个模块中的一个或多个 API 导入到当前作用域中，并分别绑定在一个变量上（在我们的例子里是 `hello`）。`module` 会将整个模块的 API 导入并绑定到一个变量上（在我们的例子里是 `foo` 和 `bar`）。`export` 会将当前模块的一个标识符（变量、函数）导出为公共 API。这些操作可以在模块定义中根据需要使用任意多次。

模块文件中的内容会被当作好像包含在作用域闭包中一样来处理，就和前面介绍的函数闭包模块一样。

5.6 小结

闭包就好像从 JavaScript 中分离出来的一个充满神秘色彩的未开化世界，只有最勇敢的人才能够到达那里。但实际上它只是一个标准，显然就是关于如何在函数作为值按需传递的词法环境中书写代码的。

当函数可以记住并访问所在的词法作用域，即使函数是在当前词法作用域之外执行，这时就产生了闭包。

如果没能认出闭包，也不了解它的工作原理，在使用它的过程中就很容易犯错，比如在循环中。但同时闭包也是一个非常强大的工具，可以用多种形式来实现模块等模式。

模块有两个主要特征：（1）为创建内部作用域而调用了一个包装函数；（2）包装函数的返回值必须至少包括一个对内部函数的引用，这样就会创建涵盖整个包装函数内部作用域的闭包。

现在我们会发现代码中到处都有闭包存在，并且我们能够识别闭包然后用它来做一些有用的事！

附录 A

动态作用域

在第 2 章中，我们对比了动态作用域和词法作用域模型，JavaScript 中的作用域就是词法作用域（事实上大部分语言都是基于词法作用域的）。

我们会简要地分析一下动态作用域，重申它与词法作用域的区别。但实际上动态作用域是 JavaScript 另一个重要机制 `this` 的表亲，本书第二部分“`this` 和对象原型”中会有详细介绍。

从第 2 章中可知，词法作用域是一套关于引擎如何寻找变量以及会在何处找到变量的规则。词法作用域最重要的特征是它的定义过程发生在代码的书写阶段（假设你没有使用 `eval()` 或 `with`）。

动态作用域似乎暗示有很好的理由让作用域作为一个在运行时就被动态确定的形式，而不是在写代码时进行静态确定的形式，事实上也是这样的。我们通过示例代码来说明：

```
function foo() {
    console.log( a ); // 2
}

function bar() {
    var a = 3;
    foo();
}

var a = 2;

bar();
```

词法作用域让 `foo()` 中的 `a` 通过 RHS 引用到了全局作用域中的 `a`，因此会输出 `2`。

而动态作用域并不关心函数和作用域是如何声明以及在何处声明的，只关心它们从何处调用。换句话说，作用域链是基于调用栈的，而不是代码中的作用域嵌套。

因此，如果 JavaScript 具有动态作用域，理论上，下面代码中的 foo() 在执行时将会输出 3。

```
function foo() {
    console.log( a ); // 3（不是 2！）
}

function bar() {
    var a = 3;
    foo();
}

var a = 2;

bar();
```

为什么会这样？因为当 foo() 无法找到 a 的变量引用时，会顺着调用栈在调用 foo() 的地方查找 a，而不是在嵌套的词法作用域链中向上查找。由于 foo() 是在 bar() 中调用的，引擎会检查 bar() 的作用域，并在其中找到值为 3 的变量 a。

很奇怪吧？现在你可能会这么想。

但这其实是因为你可能只写过基于词法作用域的代码（或者至少以词法作用域为基础进行了深入的思考），因此对动态作用域感到陌生。如果你只用基于动态作用域的语言写过代码，就会觉得这是很自然的，而词法作用域看上去才怪怪的。

需要明确的是，事实上 JavaScript 并不具有动态作用域。它只有词法作用域，简单明了。但是 this 机制某种程度上很像动态作用域。

主要区别：词法作用域是在写代码或者说定义时确定的，而动态作用域是在运行时确定的。（this 也是！）词法作用域关注函数在何处声明，而动态作用域关注函数从何处调用。

最后，this 关注函数如何调用，这就表明了 this 机制和动态作用域之间的关系多么紧密。如果想了解更多关于 this 的详细内容，参见本书第二部分“this 和对象原型”。

附录B

块作用域的替代方案

第 3 章深入研究了块作用域。至少从 ES3 发布以来，JavaScript 中就有了块作用域，而 `with` 和 `catch` 分句就是块作用域的两个小例子。

但随着 ES6 中引入了 `let`，我们的代码终于有了创建完整、不受约束的块作用域的能力。块作用域在功能上和代码风格上都拥有很多激动人心的新特性。

但如果我们想在 ES6 之前的环境中使用块作用域呢？

考虑下面的代码：

```
{
    let a = 2;
    console.log( a ); // 2
}

console.log( a ); // ReferenceError
```

这段代码在 ES6 环境中可以正常工作。但是在 ES6 之前的环境中如何才能实现这个效果？答案是使用 `catch`。

```
try{throw 2;}catch(a){
    console.log( a ); // 2
}

console.log( a ); // ReferenceError
```

天啊！这些代码既丑陋又奇怪。我们看见一个会强制抛出错误的 `try/catch`，但是它抛出

的错误就是一个值 2，然后 catch 分句中的变量声明会接收到这个值。头疼！

没错，catch 分句具有块作用域，因此它可以在 ES6 之前的环境中作为块作用域的替代方案。

“但是，”你可能会说，“鬼才要写这么丑陋的代码！”没错，没人写的代码像 CoffeeScript 编译器输出的代码，但这不是重点。

重点是工具可以将 ES6 的代码转换成能在 ES6 之前环境中运行的形式。你可以使用块作用域来写代码，并享受它带来的好处，然后在构建时通过工具来对代码进行预处理，使之可以在部署时正常工作。

事实上，这是向 ES6 中的所有（好吧，不是所有而是大部分）功能迁移的首选方式：在从 ES6 之前的环境向 ES6 过渡时，使用代码转换工具来对 ES6 代码进行处理，生成兼容 ES5 的代码。

B.1 Traceur

Google 维护着一个名为 Traceur 的项目，该项目正是用来将 ES6 代码转换成兼容 ES6 之前的环境（大部分是 ES5，但不是全部）。TC39 委员会依赖这个工具（也有其他工具）来测试他们指定的语义化相关的功能。

Traceur 会将我们的代码片段转换成什么样子？你能猜到的！

```
{
    try {
        throw undefined;
    } catch (a) {
        a = 2;
        console.log( a );
    }
}

console.log( a );
```

通过使用这样的工具，我们就可以在使用块作用域时无需考虑目标平台是否是 ES6 环境，因为 try/catch 从 ES3 开始就存在了（并且一直是这样工作的）。

B.2 隐式和显式作用域

在第 3 章中介绍块作用域时，我们的代码有一些可维护性和可扩展性方面的缺陷。有没有其他可以使用块作用域，并且还能避免这种缺陷的途径？

考虑下面这种 let 的使用方法，它被称作 let 作用域或 let 声明（对比前面的 let 定义）。

```
let (a = 2) {
    console.log( a ); // 2
}

console.log( a ); // ReferenceError
```

同隐式地劫持一个已经存在的作用域不同，let 声明会创建一个显示的作用域并与其进行绑定。显式作用域不仅更加突出，在代码重构时也表现得更加健壮。在语法上，通过强制性地将所有变量声明提升到块的顶部来产生更简洁的代码。这样更容易判断变量是否属于某个作用域。

这种模式同很多人在函数作用域中手动将 var 声明提升到函数顶部的方式很接近。let 声明有意将变量声明放在块的顶部，如果你并没有到处使用 let 定义，那么你的块作用域就很容易辨识和维护。

但是这里有一个小问题，let 声明并不包含在 ES6 中。官方的 Traceur 编译器也不接受这种形式的代码。

我们有两个选择，使用合法的 ES6 语法并且在代码规范性上做一些妥协。

```
/*let*/ { let a = 2;
    console.log( a );
}

console.log( a ); // ReferenceError
```

工具就是用来解决问题的。因此另外一个选择就是编写显式 let 声明，然后通过工具将其转换成合法的、可以工作的代码。

因此我开发了一个名为 let-er 的工具来解决这个问题。let-er 是一个构建时的代码转换器，但它唯一的作用就是找到 let 声明并对其进行转换。它不会处理包括 let 定义在内的任何其他代码。你可以安全地将 let-er 应用在 ES6 代码转换的第一步，如果有必要，接下来也可以把代码传递给 Traceur 等工具。

此外，let-er 还有一个设置项 --es6，开启它（默认是关闭的）会改变生成代码的种类。开启这个设置项时 let-er 会生成完全标准的 ES6 代码，而不会生成通过 try/catch 进行 hack 的 ES3 替代方案：

```
{
    let a = 2;
    console.log( a );
}

console.log( a ); // ReferenceError
```

因此你马上就可以在 ES6 之前的所有环境中使用 let-er，当你只关注 ES6 环境时，可以开

启设置项，这样就会生成标准的 ES6 代码。

更重要的，你甚至可以使用尚未成为 ES 官方标准的、更加好用的显式 `let` 声明。

B.3 性能

最后简单地看一下 `try/catch` 带来的性能问题，并尝试回答“为什么不直接使用 IIFE 来创建作用域”这个问题。

首先，`try/catch` 的性能的确很糟糕，但技术层面上没有合理的理由来说明 `try/catch` 必须这么慢，或者会一直慢下去。自从 TC39 支持在 ES6 的转换器中使用 `try/catch` 后，Traceur 团队已经要求 Chrome 对 `try/catch` 的性能进行改进，他们显然有很充分的动机来做这件事情。

其次，IIFE 和 `try/catch` 并不是完全等价的，因为如果将一段代码中的任意一部分拿出来用函数进行包裹，会改变这段代码的含义，其中的 `this`、`return`、`break` 和 `contine` 都会发生变化。IIFE 并不是一个普适的解决方案，它只适合在某些情况下进行手动操作。

最后问题就变成了：你是否想要块作用域？如果你想要，这些工具就可以帮助你。如果不想要，继续使用 `var` 来写代码就好了！

附录C

this词法

尽管这个标题没有详细说明 this 机制，但是 ES6 中有一个主题用非常重要的方式将 this 同词法作用域联系起来了，我们会简单地讨论一下。

ES6 添加了一个特殊的语法形式用于函数声明，叫作*箭头函数*。它看起来是下面这样的：

```
var foo = a => {
    console.log( a );
};

foo( 2 ); // 2
```

这里称作“胖箭头”的写法通常被当作*单调乏味且冗长*（挖苦）的 function 关键字的简写。

但是箭头函数除了让你在声明函数时少敲几次键盘以外，还有更重要的作用。简单来说，下面的代码有问题：

```
var obj = {
    id: "awesome",
    cool: function coolFn() {
        console.log( this.id );
    }
};

var id = "not awesome"

obj.cool(); // 酷

setTimeout( obj.cool, 100 ); // 不酷
```

问题在于 cool() 函数丢失了同 this 之间的绑定。解决这个问题有好几种办法，但最长用的就是 var self = this;。

使用起来如下所示：

```
var obj = {
    count: 0,
    cool: function coolFn() {
        var self = this;

        if (self.count < 1) {
            setTimeout( function timer(){
                self.count++;
                console.log( "awesome?" );
            }, 100 );
        }
    }
};

obj.cool(); // 酷吧?
```

var self = this 这种解决方案圆满解决了理解和正确使用 this 绑定的问题，并且没有把问题过于复杂化，它使用的是我们非常熟悉的工具：词法作用域。self 只是一个可以通过词法作用域和闭包进行引用的标识符，不关心 this 绑定的过程中发生了什么。

人们不喜欢写冗长的东西，尤其是一遍又一遍地写。因此 ES6 的一个初衷就是帮助人们减少重复的场景，事实上包括修复某些习惯用法的问题，this 就是其中一个。

ES6 中的箭头函数引入了一个叫作 this 词法的行为：

```
var obj = {
    count: 0,
    cool: function coolFn() {
        if (this.count < 1) {
            setTimeout( () => { // 箭头函数是什么鬼东西?
                this.count++;
                console.log( "awesome?" );
            }, 100 );
        }
    }
};

obj.cool(); // 很酷吧？
```

简单来说，箭头函数在涉及 this 绑定时的行为和普通函数的行为完全不一致。它放弃了所有普通 this 绑定的规则，取而代之的是用当前的词法作用域覆盖了 this 本来的值。

因此，这个代码片段中的箭头函数并非是以某种不可预测的方式同所属的 this 进行了解绑定，而只是“继承”了 cool() 函数的 this 绑定（因此调用它并不会出错）。