附录A

ECMAScript Harmony

在2004年Web开发重新焕发生机的大背景下，浏览器开发商和其他相关组织之间进行了一系列会谈，讨论应该如何改进JavaScript。ECMA-262第四版的制定工作就建立在两大相互竞争的提案基础上：一个是Netscape的JavaScript 2.0，另一个是Microsoft的JScript.NET。各方抛开在浏览器领域的竞争，聚集在ECMA麾下，提出了希望能以JavaScript为蓝本设计出一门新语言的建议方案。最初的工作草案叫做ECMAScript 4，而且很长时间以来，它好像就是JavaScript的下一个版本。后来，一个叫ECMAScript 3.1反提案的加入，令JavaScript的未来再次充满了疑问。在反复争论之后，ECMAScript 3.1成为了JavaScript的下一个版本，而且未来的工作成果——代号Harmony（和谐），将力争让ECMAScript 4向ECMAScript 3.1靠拢。

ECMAScript 3.1最终改名为ECMAScript 5，很快就完成了标准化。ECMAScript 5的详细内容本书已经介绍过了。ECMAScript 5的标准化工作一完成，Harmony立即被提上日程。Harmony与ECMAScript 5的指导思想比较一致，就是只进行增量调整，不彻底改造语言。虽然到2011年的时候，Harmony，也就是未来的ECMAScript 6，还没有全部制定完成，但其中的几个部分已经尘埃落定。本附录所要介绍的就是那些将来肯定能进入最终规范的部分。不过也提醒一下大家，在将来的实现中，这些内容的细节有可能与你在这里看到的不一样。

A.1 一般性变化

Harmony为ECMAScript引入了一些基本的变化。对这门语言来说，这些虽然不算是大的变化，但的确也弥补了它功能上的一些缺憾。

A.1.1 常量

没有正式的常量是JavaScript的一个明显缺陷。为了弥补这个缺陷，标准制定者为Harmony增加了用const关键字声明常量的语法。使用方式与var类似，但const声明的变量在初始赋值后，就不能再重新赋值了。来看一个例子。

```
const MAX_SIZE = 25;
```

可以像声明变量一样在任何地方声明常量。但在同一作用域中，常量名不能与其他变量或函数名重名，因此下列声明会导致错误：

```
const FLAG = true;
var FLAG = false; //错误!
```

除了值不能修改之外，可以像使用任何变量一样使用常量。修改常量的值，不会有任何效果，如下所示：

```
const FLAG = true;
FLAG = false;
alert(FLAG); //正确
```

支持常量的浏览器有 Firefox、Safari 3+、Opera 9+和 Chrome。在 Safari 和 Opera 中，const 与 var 的作用一样，因为前者定义的常量的值是可以修改的。

A.1.2 块级作用域及其他作用域

本书时不时就会提醒读者一句：JavaScript 没有块级作用域。换句话说，在语句块中定义的变量与在包含函数中定义的变量共享相同的作用域。Harmony 新增了定义块级作用域的语法：使用 let 关键字。

与 const 和 var 类似，可以使用 let 在任何地方定义变量并为变量赋值。区别在于，使用 let 定义的变量在定义它的代码块之外没有定义。比如说吧，下面是一个非常常见的代码块：

```
for (var i=0; i < 10; i++) {
    //执行某些操作
}

alert(i); //10
```

在上面的代码块中，变量 i 是作为代码块所在函数的局部变量来声明的。也就是说，在 for 循环执行完毕后，仍然能够读取 i 的值。如果在这里使用 let 代替 var，则循环之后，变量 i 将不复存在。看下面的例子。

```
for (let i=0; i < 10; i++) {
    //执行某些操作
}

alert(i); //错误！变量 i 没有定义
```

以上代码执行到最后一行的时候，就会出现错误，因为 for 循环一结束，变量 i 就已经没有定义了。因为不能对没有定义的变量执行操作，所以发生错误是自然的。

还有另外一种使用 let 的方式，即创建 let 语句，在其中定义只能在后续代码块中使用的变量，像下面的例子这样：

```
var num = 5;

let (num=10, multiplier=2){
    alert(num * multiplier); //20
}

alert(num); //5
```

以上代码通过 let 语句定义了一个区域，这个区域中的变量 num 等于 10，multiplier 等于 2。此时的 num 覆盖了前面用 var 声明的同名变量，因此在 let 语句块中，num 乘以 multiplier 等于 20。而出了 let 语句块之后，num 变量的值仍然是 5。这是因为 let 语句创建了自己的作用域，这个作用域里的变量与外面的变量无关。

使用同样的语法还可以创建 let 表达式，其中的变量只在表达式中有定义。再看一个例子。

```
var result = let(num=10, multiplier=2) num * multiplier;
alert(result); //20
```

这里的 let 表达式使用两个变量计算后得到一个值，保存在变量 result 中。执行表达式之后，num 和 multiplier 变量就不存在了。

在 JavaScript 中使用块级作用域，可以更精细地控制代码执行过程中变量的存废。

A.2 函数

大多数代码都是以函数方式编写的，因此 Harmony 从几个方面改进了函数，使其更便于使用。与 Harmony 中其他部分类似，对函数的改进也集中在开发人员和实现人员共同面临的难题上。

A.2.1 剩余参数与分布参数

Harmony 中不再有 arguments 对象，因此也就无法通过它来读取到未声明的参数。不过，使用剩余参数（rest arguments）语法，也能表示你期待给函数传入可变数量的参数。剩余参数的语法形式是三个点后跟一个标识符。使用这种语法可以定义可能会传进来的更多参数，然后把它们收集到一个数组中。来看一个例子。

```
function sum(num1, num2, ...nums){
    var result = num1 + num2;
    for (let i=0, len=nums.length; i < len; i++){
        result += nums[i];
    }
    return result;
}

var result = sum(1, 2, 3, 4, 5, 6);
```

以上代码定义了一个 sum() 函数，接收至少两个参数。这个函数还能接收更多参数，而其余参数都将保存在 nums 数组中。与原来的 arguments 对象不同，剩余参数都保存在 Array 的一个实例中，因此可以使用任何数组方法来操作它们。另外，即使并没有多余的参数传入函数，剩余参数对象也是 Array 的实例。

与剩余参数紧密相关的另一种参数语法是分布参数（spread arguments）。通过分布参数，可以向函数中传入一个数组，然后数组中的元素会映射到函数的每个参数上。分布参数的语法形式与剩余参数的语法相同，就是在值的前面加三个点。唯一的区别是分布参数在调用函数的时候使用，而剩余参数在定义函数的时候使用。比如，我们可以不给 sum() 函数一个一个地传入参数，而是传入分布参数：

```
var result = sum(...[1, 2, 3, 4, 5, 6]);
```

在这里，我们将一个数组作为分布参数传给了 sum() 函数。以上代码在功能上与下面这行代码等价：

```
var result = sum.apply(this, [1, 2, 3, 4, 5, 6]);
```

A.2.2 默认参数值

ECMAScript 函数中的所有参数都是可选的，因为实现不会检查传入的参数数量。不过，除了手工

检查传入了哪个参数之外，你还可以为参数指定默认值。如果调用函数时没有传入该参数，那么该参数就会使用默认值。

要为参数指定默认值，可以在参数名后面直接加上等于号和默认值，就像下面这样：

```
function sum(num1, num2=0){
    return num1 + num2;
}

var result1 = sum(5);
var result2 = sum(5, 5);
```

这个 sum()函数接收两个参数，但第二个参数是可选的，因为它的默认值为 0。使用可选参数的好处是开发人员不用再去检查是否给某个参数传入了值，如果没有的话就使用某个特定的值。默认参数值帮你解除了这个困扰。

A.2.3 生成器

所谓生成器，其实就是一个对象，它每次能生成一系列值中的一个。对 Harmony 而言，要创建生成器，可以让函数通过 yield 操作符返回某个特殊的值。对于使用 yield 操作符返回值的函数，调用它时就会创建并返回一个新的 Generator 实例。然后，在这个实例上调用 next()方法就能取得生成器的第一个值。此时，执行的是原来的函数，但执行流到 yield 语句就会停止，只返回特定的值。从这个角度看，yield 与 return 很相似。如果再次调用 next()方法，原来函数中位于 yield 语句后的代码会继续执行，直到再次遇见 yield 语句时停止执行，此时再返回一个新值。来看下面的例子。

```
function myNumbers(){
    for (var i=0; i < 10; i++){
        yield i * 2;
    }
}

var generator = myNumbers();

try {
    while(true){
        document.write(generator.next() + "<br />");
    }
} catch(ex){
    //有意没有写代码
} finally {
    generator.close();
}
```

调用 myNumbers()函数后，会得到一个生成器。myNumbers()函数本身非常简单，包含一个每次循环都产生一个值的 for 循环。每次调用 next()方法都会执行一次 for 循环，然后返回下一个值。第一个值是 0，第二个值是 2，第三个值是 4，依此类推。在 myNumbers()函数完成退出而没有执行 yield 语句时（最后一次循环判断 i 不小于 10 的时候），生成器会抛出 StopIteration 错误。因此，为了输出生成器能产生的所有数值，这里用一个 try-catch 结构包装了一个 while 循环，以避免出错时中断代码执行。

如果不再需要某个生成器，最好是调用它的 close()方法。这样会执行原始函数的其他部分，包括 try-catch 相关的 finally 语句块。

在需要一系列值，而每一个值又与前一个值存在某种关系的情况下，可以使用生成器。

A.3 数组及其他结构

Harmony 的另一个重点是数组。数组是 JavaScript 使用最频繁的一种数据结构，因此定义一些更直观更方便地使用数组的方式，绝对是改进这门语言时最优先考虑的事。

A.3.1 迭代器

迭代器也是一个对象，它能迭代一系列值并每次返回其中一个值。想象一下使用 for 或 for-in 循环，这时候就是在迭代一批值，而且每次操作其中的一个值。迭代器的作用相同，只不过用不着使用循环了。Harmony 为各种类型的对象都定义了迭代器。

要为对象创建迭代器，可以调用 Iterator 构造函数，传入想要迭代其值的对象。要取得对象中的下一个值，可以调用迭代器的 next()方法。默认情况下，这个方法会返回一个数组。如果迭代的是数组，那么返回数组的第一个元素是值的索引，如果迭代的是对象，那么返回数组的第一个元素是值的属性名；返回数组的第二个元素是值本身。如果所有值都已经迭代了一遍，则再调用 next()会抛出 StopIteration 错误。看下面这个例子。

```
var person = {
    name: "Nicholas",
    age: 29
};
var iterator = new Iterator(person);

try {
    while(true){
        let value = iterator.next();
        document.write(value.join(":") + "<br>");
    }
} catch(ex){
    //有意没有写代码
}
```

以上代码为 person 对象创建了一个迭代器。第一次调用 next()方法，返回数组["name", "Nicholas"]，第二次调用返回数组["age", 29]。以上代码的输入结果为：

```
name:Nicholas
age:29
```

如果为非数组对象创建迭代器，则迭代器会按照与使用 for-in 循环一样的顺序，返回对象的每个属性。这就意味着迭代器也只能返回对象的实例属性，而且返回属性的顺序也会因实现而异。为数组创建的迭代器也类似，即按数组元素顺序依次返回值，下面是一个例子。

```
var colors = ["red", "green", "blue"];
var iterator = new Iterator(colors);

try {
    while(true){
        let value = iterator.next();
        document.write(value.join(":") + "<br>");
```

```
    }
} catch(ex){

}
```

以上代码的输出结果如下：

```
0:red
1:green
2:blue
```

如果你只想让next()方法返回对象的属性名或者数组的索引值，可以在创建迭代器时为Iterator构造函数传入第二个参数true，如下所示：

```
var iterator = new Iterator(colors, true);
```

在这样创建的迭代器上每次调用next()方法，只会返回数组中每个值的索引，而不会返回包含索引和值数组。

如果想为自定义类型创建迭代器，需要定义一个特殊的方法__iterator__()，这个方法应该返回一个包含next()方法的对象。当把自定义类型传给Iterator构造函数时，就会调用那个特殊的方法。

A.3.2 数组领悟

所谓数组领悟（array comprehensions），指的是用一组符合某个条件的值来初始化数组。Harmony定义的这项功能借鉴了Python中流行的一个语言结构。JavaScript中数组领悟的基本形式如下：

```
array = [ value for each (variable in values) condition ];
```

其中，value是实际会包含在数组中的值，它源自values数组。for each结构会循环values中的每一个值，并将每个值保存在变量variable中。如果保存在variable中的值符合condition条件，就会将这个值添加到结果数组中。下面是一个例子。

```
//原始数组
var numbers = [0,1,2,3,4,5,6,7,8,9,10];

//把所有元素复制到新数组
var duplicate = [i for each (i in numbers)];

//只把偶数复制到新数组
var evens = [i for each (i in numbers) if (i % 2 == 0)];

//把每个数乘以2后的结果放到新数组中
var doubled = [i*2 for each (i in numbers)];

//把每个奇数乘以3后的结果放到新数组中
var tripledOdds = [i*3 for each (i in numbers) if (i % 2 > 0)];
```

在以上代码的数组领悟部分，我们使用变量i迭代了numbers中的所有值，而其中一些语句给出了条件，以筛选最终包含在数组中的结果。本质上讲，只要条件求值为true，该值就会添加到数组中。与自己编写同样功能的for循环相比，数组领悟的语法稍有不同，但却更加简洁。Firefox 2+是唯一支持数

组领悟的浏览器，而且要使用这个功能，必须将<script>的 type 属性值指定为"application/javascript;version=1.7"。

数组领悟语法的 values 也可以是一个生成器或者一个迭代器。

A.3.3 解构赋值

从一组值中挑出一或多个值，然后把它们分别赋给独立的变量，这也是一个很常见的需求。就拿迭代器的 next()方法返回的数组来说，假设这个数组包含着对象中一个属性的名称和值。为了把这个属性和值分别保存在各自的变量中，需要写两个语句，如下所示。

```
var nextValue = ["color", "red"];
var name = nextValue[0];
var value = nextValue[1];
```

而使用解构赋值（destructuring assignments）语法，用一条语句即可解决问题：

```
var [name, value] = ["color", "red"];
alert(name);     //"color"
alert(value);    //"red"
```

在传统的 JavaScript 中，数组字面量是不能出现在等于号（赋值操作符）左边的。解构赋值的这种语法表示的是把等于号右边数组中包含的值，分别赋给等于号左边数组中的变量。结果就是变量 name 的值为"color"，变量 value 的值为"red"。

如果你不想取得数组中所有的值，可以只在数组字面量中给出对应的变量，比如：

```
var [, value] = ["color", "red"];
alert(value);   //"red"
```

这样就只会给变量 value 赋值，值为"red"。

有了解构赋值，还可做点有创意的事儿，比如交换变量的值。在 ECMAScript 3 中，要交换两个变量的值，一般是要这样写代码的：

```
var value1 = 5;
var value2 = 10;

var temp = value1;
value1 = value2;
value2 = temp;
```

利用解构后的数组赋值，可以省掉那个临时变量 temp，比如：

```
var value1 = 5;
var value2 = 10;

[value2, value1] = [value1, value2];
```

解构赋值同样适用于对象，看下面这个例子：

```
var person = {
    name: "Nicholas",
    age: 29
};
```

```
var { name: personName, age: personAge } = person;

alert(personName); //"Nicholas"
alert(personAge);  //29
```

与使用数组字面量一样，看到等于号左边出现了对象字面量，那就是解构赋值表达式。这条语句实际上定义了两个变量，personName 和 personAge，它们分别得到了 person 对象中对应的值。与数组解构赋值一样，在对象解构赋值中也可以选择要取得的值，比如：

```
var { age: personAge } = person;
alert(personAge); //29
```

以上代码只取得了 person 对象中 age 属性的值，将它赋给了变量 personAge。

A.4 新对象类型

Harmony 为 JavaScript 定义了几个新的对象类型。这几个新类型提供了以前只有 JavaScript 引擎才能使用的功能。

A.4.1 代理对象

Harmony 为 JavaScript 引入了代理的概念。所谓代理（proxy），就是一个表示接口的对象，对它的操作不一定作用在代理对象本身。举个例子，设置代理对象的一个属性，实际上可能会在另一个对象上调用一个函数。代理是一种非常有用的抽象机制，能够通过 API 只公开部分信息，同时还能对数据源进行全面控制。

要创建代理对象，可以使用 Proxy.create()方法，传入一个 handler（处理程序）对象和一个可选的 prototype（原型）对象：

```
var proxy = Proxy.create(handler);

//创建一个以 myObject 为原型的代理对象
var proxy = Proxy.create(handler, myObject);
```

其中，handler 对象包含用于定义捕捉器（trap）的属性。捕捉器本身是函数，用于处理（捕捉）原生功能，以便该功能能够以另一种方式来处理。要确保代理对象能够按照预期工作，至少要实现以下 7 种基本的捕捉器。

- getOwnPropertyDescriptor：当在代理对象上调用 Object.getOwnPropertyDescriptor()时调用的函数。这个函数以接收到的属性名作为参数，返回属性描述符，或者在属性不存在时返回 null。
- getPropertyDescriptor：当在代理对象上调用 Object.getPropertyDescriptor()时调用的函数。（这是 Harmony 中的新方法。）这个函数以接收到的属性名作为参数，返回属性描述符，或者在属性不存在时返回 null。
- getOwnPropertyNames：当在代理对象上调用 Object.getOwnPropertyNames ()时调用的函数。这个函数以接收到的属性名作为参数，应该返回一个字符串数组。
- getPropertyNames：当在代理对象上调用 Object.getPropertyNames ()时调用的函数。（这是 Harmony 中的新方法。）这个函数以接收到的属性名作为参数，应该返回一个字符串数组。

- defineProperty：当在代理对象上调用 Object.defineProperty()时调用的函数。这个函数以接收到的属性名和属性描述符作为参数。
- delete：定义在对象属性上使用 delete 操作符时调用的函数。属性名以参数形式传进来，如果删除成功则返回 true，删除失败返回 false。
- fix：当调用 Object.freeze()、Object.seal()或 Object.preventExtensions()时调用的函数。当在代理对象上调用这几个方法时，返回 undefined 以抛出错误。

除了这 7 个基本的捕捉器，还有 6 个派生的捕捉器（derived trap）。与基本捕捉器不同，少定义一个或几个派生捕捉器不会导致错误。每个派生的捕捉器都会覆盖一种默认的 JavaScript 行为。

- has 在对象上使用 in 操作符（例如"name" in object）时调用的函数。以接收到的属性名作为参数，返回 true 表示对象包含该属性，否则返回 false。
- hasOwn：在代理对象上调用 hasOwnProperty()方法时调用的函数。以接收到的属性名作为参数，返回 true 表示对象包含该属性，否则返回 false。
- get：在读取属性时调用的函数。这个函数接收两个参数，即包含被读属性的对象的引用及属性名。这个对象引用可能是代理对象本身，也可能是继承了代理对象的对象。
- set：在写入属性时调用的函数。这个函数接收三个参数，即包含被写属性的对象的引用、属性名和属性值。与 get 类似，这个对象引用可能是代理对象本身，也可能是继承了代理对象的对象。
- enumerate：当代理对象被放在 for-in 循环中时调用的函数。这个函数必须返回一个字符串数组，其中包含在 for-in 循环中使用的相应属性名。
- keys：当在代理对象上调用 Object.keys()时调用的函数。与 enumerate 类似，这个函数也必须返回一个字符串数组。

在需要公开 API，而同时又要避免使用者直接操作底层数据的时候，可以使用代理。例如，假设你想实现一个传统的栈数据类型。虽然数组可以作为栈来使用，但你想保证人们只使用 push()、pop()和 length。在这种情况下，就可以基于数组创建一个代理对象，只对外公开这三个对象成员。

```
/*
 *实验 ES 6 代理对象。这个实验在数组的基础上创建一个栈数据结构。
 *代理在此用于从接口过滤"push"、"pop"和"length"之外的成员，让数组成为一个纯粹的栈，
 *任何人不能直接操作其内容。
 */

var Stack = (function(){

    var stack = [],
        allowed = ["push", "pop", "length" ];

    return Proxy.create({
        get: function(receiver, name){;
            if (allowed.indexOf(name) > -1){
                if(typeof stack[name] == "function"){
                    return stack[name].bind(stack);
                } else {
                    return stack[name];
                }
            } else {
                return undefined;
            }
        }
```

```
    });

});

var mystack = new Stack();

mystack.push("hi");
mystack.push("goodbye");

console.log(mystack.length);   //1

console.log(mystack[0]);       //未定义
console.log(mystack.pop());    //"goodbye"
```

以上代码创建了一个构造函数 Stack。但它没有使用 this，而是返回了一个对数组操作进行包装的代理对象。这个代理对象只定义了一个 get 捕捉器，该函数检测了一组允许的属性，然后才返回相应的值。如果引用的是不被允许的属性，那么捕捉器就返回 undefined；如果引用的是 push()、pop() 和 length，则一切正常。这里的关键是 get 捕捉器，它根据允许的成员过滤了对象的成员。如果该成员是函数，就返回一个与底层数组对象绑定的函数，这样操作针对的就是数组而非代理对象。

A.4.2 代理函数

除了创建代理对象之外，Harmony 还支持创建**代理函数**（proxy function）。代理函数与代理对象的区别是它可以执行。要创建代理函数，可以调用 Proxy.createFunction()方法，传入一个 handler（处理程序）对象、一个调用捕捉器函数和一个可选的构造函数捕捉器函数。例如：

```
var proxy = Proxy.createFunction(handler, function(){}, function(){});
```

与代理对象一样，handler 对象也有同样多的捕捉器。调用捕捉器函数是在代理函数执行（如 proxy()）时运行的代码。构造函数捕捉器是在用 new 操作符调用代理函数（如 new proxy()）时运行的代码。如果没有指定构造函数捕捉器，则使用调用捕捉器作为构造函数。

A.4.3 映射与集合

Map 类型，也称为**简单映射**，只有一个目的：保存一组键值对儿。开发人员通常都使用普通对象来保存键值对儿，但问题是那样做会导致键容易与原生属性混淆。简单映射能做到键和值与对象属性分离，从而保证对象属性的安全存储。以下是使用简单映射的几个例子。

```
var map = new Map();

map.set("name", "Nicholas");
map.set("book", "Professional JavaScript");

console.log(map.has("name")); //true
console.log(map.get("name")); //"Nicholas"

map.delete("name");
```

简单映射的基本 API 包括 get()、set()和 delete()，每个方法的作用看名字就知道了。键可以是原始值，也可是引用值。

与简单映射相关的是 Set 类型。集合就是一组不重复的元素。与简单映射不同的是，集合中只有键，

没有与键关联的值。在集合中，添加元素要使用 add()方法，检查元素是否存在要使用 has()方法，而删除元素要使用 delete()方法。以下是基本的使用示例。

```
var set = new Set();
set.add("name");

console.log(set.has("name")); //true
set.delete("name");

console.log(set.has("name")); //false
```

截止到 2011 年 10 月，规范中关于 Map 和 Set 的内容还没有最后定稿。因此，在 JavaScript 引擎实现该规范时，有些细节可能会发生变化。

A.4.4 WeakMap

WeakMap 是 ECMAScript 中唯一一个能让你知道什么时候对象已经完全解除引用的类型。WeakMap 与简单映射很相似，也是用来保存键值对儿的。它们的主要区别在于，WeakMap 的键必须是对象，而在对象已经不存在时，相关的键值对儿就会从 WeakMap 中被删除。例如：

```
var key = {},
    map = new WeakMap();

map.set(key, "Hello!");

//解除对键的引用，从而删除该值
key = null;
```

至于什么情况下适合使用 WeakMap，目前还不清楚。不过，Java 中倒是有一个相同的数据结构叫 WeakHashMap；于是，JavaScript 又多了一种数据类型。

A.4.5 StructType

JavaScript 一个最大的不足是使用一种数据类型表示所有数值。WebGL 为解决这个问题引入了类型化数组，而 ECMAScript 6 则引入了类型化结构，为这门语言带来了更多的数值数据类型。结构类型（StructType）与 C 语言中的结构类似；在 C 语言中，可以把多个属性组合成一条记录。对于 JavaScript 的结构类型，通过指定属性及其保存的数据类型，也可以创建类似的数据结构。早期的实现定义了以下几种块类型。

- uint8：无符号 8 位整数。
- int8：有符号 8 位整数。
- uint16：无符号 16 位整数。
- int16：有符号 16 位整数。
- uint32：无符号 32 位整数。
- int32：有符号 32 位整数。
- float32：32 位浮点数。
- float64：64 位浮点数。

这些块类型都只能包含一个值。将来还有望在这 8 种类型基础上进一步扩展。

要创建结构类型的对象，可以使用 new 操作符调用 StructType，传入对象字面量形式的属性定义。

```
var Size = new StructType({ width: uint32, height: uint32 });
```

以上代码创建了一个名为 Size 的新结构类型，该类型带有两个属性：width 和 height。这两个属性都应该保存无符号 32 位整数。而变量 Size 实际上是一个构造函数，可以像使用对象的构造函数一样使用它。要实例化这个结构类型，需要向构造函数中传入一个带属性值的对象字面量。

```
var boxSize = new Size({ width: 80, height: 60 });
console.log(boxSize.width);  //80
console.log(boxSize.height); //60
```

这样，就创建了 Size 的一个宽为 80、高为 60 的实例。实例的属性可以被读写，但始终都必须包含 32 位无符号整数。

将属性定义为另一个结构类型，可以得到更复杂的结构类型。例如：

```
var Location = new StructType({ x: int32, y: int32 });
var Box = new StructType({ size: Size, location: Location });

var boxInfo = new Box({ size: { width:80, height:60 }, location: { x: 0, y: 0 }});
console.log(boxInfo.size.width); //80
```

这个例子创建了一个简单的结构类型 Location，又创建了一个复杂的结构类型 Box。Box 的属性本身也是结构类型。Box 构造函数仍然接收对象字面量参数，以便为每个属性定义值，但它会检查传入值的数据类型，以确保作为属性值的数据类型正确。

A.4.6 ArrayType

与结构类型密切相关的是数组类型。通过数组类型（ArrayType）可以创建一个数组，并限制数组的值必须是某种特定的类型（与 WebGL 中的类型化数组很相似）。要创建新的数组类型，可以调用 ArrayType 构造函数，并传入它应该保存的数据类型以及应该保存的元素数目。例如：

```
var SizeArray = new ArrayType(Size, 2);
var boxes = new BoxArray([ { width: 80, height: 60 }, { width: 50, height: 50 } ]);
```

以上代码创建了一个名为 SizeArray 的数组类型，这个数组类型只能保存 Size 的实例，同时也给数组分配了两个该实例的位置。要实例化数组类型，可以传入一个数组，其中包含应该转换的数据。数据可以是字面量，只要该字面量能提升为正确的数据类型即可（比如在这个例子中，传入的字面量可以提升为结构类型）。

A.5 类

开发人员一直吵着要在 JavaScript 中增加一种语法，用于定义类似于 Java 的类。ECMAScript 6 最终确实定义了这种语法。但 JavaScript 中的类只是一种语法糖，覆盖在目前基于构造函数和基于原型的方式和类型之上。先看看下面的类型定义。

```
function Person(name, age){
    this.name = name;
    this.age = age;
}

Person.prototype.sayName = function(){
    alert(this.name);
```

```
};

Person.prototype.getOlder = function(years){
    this.age += years;
};
```

再看看使用新语法定义的类：

```
class Person {

    constructor(name, age){
        public name = name;
        public age = age;
    }

    sayName(){
        alert(this.name);
    }

    getOlder(years){
        this.age += years;
    }

}
```

新语法以关键字 class 开头，然后就是类型名，而花括号中定义的是属性和方法。定义方法不必再使用 function 关键字，有方法名和圆括号就可以。如果把方法命名为 constructor，那它就是这个类的构造函数（与前一个例子中的 Person 函数一样）。在这个类中定义的方法和属性都会添加到原型上，具体来说，sayName()和 getOlder()都是在 Person.prototype 上定义的。

在构造函数中，public 和 private 关键字用于创建对象的实例属性。这个例子中的 name 和 age 都是公有属性。

A.5.1 私有成员

关于类语法的建议是默认支持私有成员的，包括实例中的私有成员和原型中的私有成员。private 关键字表示成员是私有的，不能在类方法之外访问。要访问私有成员，可以使用一种特殊的语法，即调用 private()函数并传入 this 对象，然后再访问私有成员。例如，下面这个例子把 Person 类的 age 改成为私有属性：

```
class Person {

    constructor(name, age){
        public name = name;
        private age = age;
    }

    sayName(){
        alert(this.name);
    }

    getOlder(years){
        private(this).age += years;
    }

}
```

这种用于访问私有成员的语法还没有定论，将来很可能会改变。

A.5.2 getter 和 setter

新的类语法支持直接为属性定义 getter 和 setter，从而避免了调用 Object.defineProperty() 的麻烦。为属性定义 getter 和 setter 与定义方法类似，只不过要在方法名前加上 get 和 set 关键字。例如：

```
class Person {

    constructor(name, age){
        public name = name;
        public age = age;
        private innerTitle = "";

        get title(){
            return innerTitle;
        }

        set title(value){
            innerTitle = value;
        }
    }
    sayName(){
        alert(this.name);
    }

    getOlder(years){
        this.age += years;
    }
}
```

这个 Person 类为 title 属性定义了一个 getter 和一个 setter。这两个操作 innerTitle 变量的函数都定义在了构造函数中。要为原型属性定义 getter 和 setter，语法相同，但要在构造函数外部定义。

A.5.3 继承

使用类语法而不是过去那种 JavaScript 语法，最大的好处是容易实现继承。有了类语法，只要使用与其他语言相同的 extends 关键字就能实现继承，而不必去考虑借用构造函数或者原型连缀。例如：

```
class Employee extends Person {
    constructor(name, age){
        super(name,age);
    }
}
```

以上代码创建了一个新类 Employee，它继承了 Person 类。在简单的语法背后，已经自动实现了原型连缀，而且通过使用 super() 函数，也正式支持了借用构造函数。从逻辑上看，上面的代码与下面的代码是等价的：

```
function Employee(name, age){
    Person.call(this, name, age);
}
```

```
Employee.prototype = new Person();
```

除了这种风格的继承，类语法还允许直接将对象指定为其原型，方法就是用 prototype 关键字代替 extends：

```
var basePerson = {
    sayName: function(){
        alert(this.name);
    },

    getOlder: function(years){
        this.age += years;
    }
};

class Employee prototype basePerson {
    constructor(name, age){
        public name = name;
        public age = age;
    }
}
```

这个例子将 basePerson 对象直接指定为 Employee.prototype，从而实现了与目前使用 Object.create()实现的一样的继承。

A.6 模块

模块（或者“命名空间”、“包”）是组织 JavaScript 应用代码的重要方法。每个模块都包含着独立于其他模式的特定、独一无二的功能。JavaScript 开发中曾出现过一些临时性的模块格式，而 ECMAScript 6 则对如何创建和管理模块给出了标准的定义。

模块在其自己的顶级执行环境中运行，因而不会污染导入它的全局执行环境。默认情况下，模块中声明的所有变量、函数、类等都是该模块私有的。对于应该向外部公开的成员，可以在前面加上 export 关键字。例如：

```
module MyModule {
    //公开这些成员
    export let myobject = {};
    export function hello(){ alert("hello"); };

    //隐藏这些成员
    function goodbye(){
        //...
    }
}
```

这个模块公开了一个名为 myobject 的对象和一个名为 hello()的函数。可以在页面或其他模块中使用这个模块，也可以只导入模块中的一个成员或者两个成员。导入模块要使用 import 命令：

```
//只导入 myobject
import myobject from MyModule;
console.log(myobject);

//导入所有公开的成员
import * from MyModule;
```

```
console.log(myobject);
console.log(hello);

//列出要导入的成员名
import {myobject, hello} from MyModule;
console.log(myobject);
console.log(hello);

//不导入，直接使用模块
console.log(MyModule.myobject);
console.log(MyModule.hello);
```

在执行环境能够访问到模块的情况下，可以直接调用模块中对外公开的成员。导入操作只不过是把模块中的个别成员拿到当前执行环境中，以便直接操作而不必引用模块。

外部模块

通过提供模块所在外部文件的 URL，也可以动态加载和导入模块。为此，首先要在模块声明后面加上外部文件的 URL，然后再导入模块成员：

```
module MyModule from "mymodule.js";
import myobject from MyModule;
```

以上声明会通知 JavaScript 引擎下载 `mymodule.js` 文件，然后从中加载名为 `MyModule` 的模块。请读者注意，这个调用会阻塞进程。换句话说，JavaScript 引擎在下载完外部文件并对其求值之前，不会处理后面的代码。

如果你只想包含模块中对外公开的某些成员，不想把整个模块都加载进来，可以像下面这样使用 `import` 指令：

```
import myobject from "mymodule.js";
```

总之，模块就是一种组织相关功能的手段，而且能够保护全局作用域不受污染。

附录 B

严格模式

ECMAScript 5 最早引入了“严格模式”（strict mode）的概念。通过严格格式，可以在函数内部选择进行较为严格的全局或局部的错误条件检测。使用严格模式的好处是可以提早知道代码中存在的错误，及时捕获一些可能导致编程错误的 ECMAScript 行为。

理解严格模式的规则非常重要，ECMAScript 的下一个版本将以严格模式为基础制定。支持严格模式的浏览器包括 IE10+、Firefox 4+、Safari 5.1+和 Chrome。

B.1 选择使用

要选择进入严格模式，可以使用严格模式的**编译指示**（pragma），实际上就是一个不会赋给任何变量的字符串：

```
"use strict";
```

这种语法（从 ECMAScript 3 开始支持）可以向后兼容那些不支持严格模式的 JavaScript 引擎。支持严格模式的引擎会启动这种模式，而不支持该模式的引擎就当遇到了一个未赋值的字符串字面量，会忽略这个编译指示。

如果是在全局作用域中（函数外部）给出这个编译指示，则整个脚本都将使用严格模式。换句话说，如果把带有这个编译指示的脚本放到其他文件中，则该文件中的 JavaScript 代码也将处于严格模式下。

也可以只在函数中打开严格模式，就像下面这样：

```
function doSomething(){
    "use strict";

    //其他代码
}
```

如果你没有控制页面中所有脚本的权力，建议只在需要测试的特定函数中开启严格模式。

B.2 变量

在严格模式下，什么时候创建变量以及怎么创建变量都是有限制的。首先，不允许意外创建全局变量。在非严格模式下，可以像下面这样创建全局变量：

```
//未声明变量
//非严格模式：创建全局变量
//严格模式：抛出 ReferenceError
message = "Hello world! ";
```

即使 message 前面没有 var 关键字，即使没有将它定义为某个全局对象的属性，也能将 message 创建为全局变量。但在严格模式下，如果给一个没有声明的变量赋值，那代码在执行时就会抛出 ReferenceError。

其次，不能对变量调用 delete 操作符。非严格模式允许这样操作，但会静默失败（返回 false）。而在严格模式下，删除变量也会导致错误。

```
//删除变量
//非严格模式：静默失败
//严格模式：抛出 ReferenceError

var color = "red";
delete color;
```

严格模式下对变量也有限制。特别地，不能使用 implements、interface、let、package、private、protected、public、static 和 yield 作为变量名。这些都是保留字，将来的 ECMAScript 版本中可能会用到它们。在严格模式下，用以上标识符作为变量名会导致语法错误。

B.3 对象

在严格模式下操作对象比在非严格模式下更容易导致错误。一般来说，非严格模式下会静默失败的情形，在严格模式下就会抛出错误。因此，在开发中使用严格模式会加大早发现错误的可能性。

在下列情形下操作对象的属性会导致错误：

- 为只读属性赋值会抛出 TypeError；
- 对不可配置的（nonconfigurable）的属性使用 delete 操作符会抛出 TypeError；
- 为不可扩展的（nonextensible）的对象添加属性会抛出 TypeError。

使用对象的另一个限制与通过对象字面量声明对象有关。在使用对象字面量时，属性名必须唯一。例如：

```
//重名属性
//非严格模式：没有错误，以第二个属性为准
//严格模式：抛出语法错误

var person = {
                name: "Nicholas",
                name: "Greg"
             };
```

这里的对象 person 有两个属性，都叫 name。在非严格模式下，person 对象的 name 属性值是第二个，而在严格模式下，这样的代码会导致语法错误。

B.4 函数

首先，严格模式要求命名函数的参数必须唯一。以下面这个函数为例：

```
//重名参数
//非严格模式：没有错误，只能访问第二个参数
//严格模式：抛出语法错误
```

```
function sum (num, num){
    //do something
}
```

在非严格模式下，这个函数声明不会抛出错误。通过参数名只能访问第二个参数，要访问第一个参数必须通过 arguments 对象。

在严格模式下，arguments 对象的行为也有所不同。在非严格模式下，修改命名参数的值也会反映到 arguments 对象中，而严格模式下这两个值是完全独立的。例如：

```
//修改命名参数的值
//非严格模式：修改会反映到 arguments 中
//严格模式：修改不会反映到 arguments 中

function showValue(value){
    value = "Foo";
    alert(value);          //"Foo"
    alert(arguments[0]); //非严格模式："Foo"
                          //严格模式："Hi"
}

showValue("Hi");
```

以上代码中，函数 showValue()只有一个命名参数 value。调用这个函数时传入了一个参数"Hi"，这个值赋给了 value。而在函数内部，value 被改为"Foo"。在非严格模式下，这个修改也会改变 arguments[0]的值，但在严格模式下，arguments[0]的值仍然是传入的值。

另一个变化是淘汰了 arguments.callee 和 arguments.caller。在非严格模式下，这两个属性一个引用函数本身，一个引用调用函数。而在严格模式下，访问哪个属性都会抛出 TypeError。例如：

```
//访问 arguments.callee
//非严格模式：没有问题
//严格模式：抛出 TypeError

function factorial(num){
    if (num <= 1) {
        return 1;
    } else {
        return num * arguments.callee(num-1)
    }
}

var result=factorial(5);
```

类似地，尝试读写函数的 caller 属性，也会导致抛出 TypeError。所以，对于上面的例子而言，访问 factorial.caller 也会抛出错误。

与变量类似，严格模式对函数名也做出了限制，不允许用 implements、interface、let、package、private、protected、public、static 和 yield 作为函数名。

对函数的最后一点限制，就是只能在脚本的顶级和在函数内部声明函数。也就是说，在 if 语句中声明函数会导致语法错误：

```
//在 if 语句中声明函数
//非严格模式：将函数提升到 if 语句外部
//严格模式：抛出语法错误

if (true){
    function doSomething(){
        //...
    }
}
```

在非严格模式下，以上代码能在所有浏览器中运行，但在严格模式下会导致语法错误。

B.5 eval()

饱受诟病的 eval()函数在严格模式下也得到了提升。最大的变化就是它在包含上下文中不再创建变量或函数。例如：

```
//使用 eval()创建变量
//非严格模式：弹出对话框显示 10
//严格模式：调用 alert(x)时会抛出 ReferenceError

function doSomething(){
    eval("var x=10");
    alert(x);
}
```

如果是在非严格模式下，以上代码会在函数 doSomething()中创建一个局部变量 x，然后 alert()还会显示该变量的值。但在严格模式下，在 doSomething()函数中调用 eval()不会创建变量 x，因此调用 alert()会导致抛出 ReferenceError，因为 x 没有定义。

可以在 eval()中声明变量和函数，但这些变量或函数只能在被求值的特殊作用域中有效，随后就将被销毁。因此，以下代码可以运行，没有问题：

```
"use strict";
var result = eval("var x=10, y=11; x+y");
alert(result); //21
```

这里在 eval()中声明了变量 x 和 y，然后将它们加在一起，返回了它们的和。于是，result 变量的值是 21，即 x 和 y 相加的结果。而在调用 alert()时，尽管 x 和 y 已经不存在了，result 变量的值仍然是有效的。

B.6 eval 与 arguments

严格模式已经明确禁止使用 eval 和 arguments 作为标识符，也不允许读写它们的值。例如：

```
//把 eval 和 arguments 作为变量引用
//非严格模式：没问题，不出错
//严格模式：抛出语法错误

var eval = 10;
var arguments = "Hello world!";
```

在非严格模式下，可以重写 eval，也可以给 arguments 赋值。但在严格模式下，这样做会导致语法错误。不能将它们用作标识符，意味着以下几种使用方式都会抛出语法错误：

- 使用 var 声明；
- 赋予另一个值；
- 尝试修改包含的值，如使用++；
- 用作函数名；
- 用作命名的函数参数；
- 在 try-catch 语句中用作例外名。

B.7 抑制 this

JavaScript 中一个最大的安全问题，也是最容易让人迷茫的地方，就是在某些情况下如何抑制 this 的值。在非严格模式下使用函数的 apply()或 call()方法时，null 或 undefined 值会被转换为全局对象。而在严格模式下，函数的 this 值始终是指定的值，无论指定的是什么值。例如：

```
//访问属性
//非严格模式：访问全局属性
//严格模式：抛出错误，因为 this 的值为 null

var color = "red";

function displayColor(){
    alert(this.color);
}

displayColor.call(null);
```

以上代码向 displayColor.call()中传入了 null，如果在是非严格模式下，这意味着函数的 this 值是全局对象。结果就是弹出对话框显示"red"。而在严格模式下，这个函数的 this 的值是 null，因此在访问 null 的属性时就会抛出错误。

B.8 其他变化

严格模式还有其他一些变化，希望读者也能留意。首先是抛弃了 with 语句。非严格模式下的 with 语句能够改变解析标识符的路径，但在严格模式下，with 被简化掉了。因此，在严格模式下使用 with 会导致语法错误。

```
//with 的语句用法
//非严格模式：允许
//严格模式：抛出语法错误

with(location){
    alert(href);
}
```

严格模式也去掉了 JavaScript 中的八进制字面量。以 0 开头的八进制字面量过去经常会导致很多错误。在严格模式下，八进制字面量已经成为无效的语法了。

```
//使用八进制字面量
//非严格模式：值为 8
//严格模式：抛出语法错误

var value = 010;
```

本书前面提到过，ECMAScript 5 也修改了严格模式下 parseInt() 的行为。如今，八进制字面量在严格模式下会被当作以 0 开头的十进制字面量。例如：

```
//使用 parseInt() 解析八进制字面量
//非严格模式：值为 8
//严格模式：值为 10

var value = parseInt("010");
```

附录 C

JavaScript 库

JavaScript 库可以帮助我们跨越浏览器差异的鸿沟，并对复杂的浏览器功能提供更为简便的访问方式。程序库有两种形式：通用库和专用库。通用 JavaScript 库提供了对常见浏览器功能的访问，可以作为网站或者 Web 应用的基础。专用库则只做特定的事，仅用于网站或者 Web 应用的某些部分。本附录给出了这些库与其功能的概况，并提供了相关网站作为你的参考资源。

C.1 通用库

通用 JavaScript 库提供横跨几个主题的功能。所有的通用库都尝试通过使用新 API 包装常见功能来统一浏览器的接口、减小实现差异。某些 API 看上去与原生功能很相似，而另一些则完全不同。通用库一般提供与 DOM 交互的功能、支持 Ajax、同时还有一些协助常见任务的工具方法。

C.1.1 YUI

它是一个开源 JavaScript 与 CSS 库，以一种组件方式设计的。这个库不只有一个文件；它包含了很多文件，提供各种不同的配置，让你可以按需载入。YUI（Yahoo!User Interface Library，雅虎用户界面库）涵盖了 JavaScript 的所有方面，从基本的工具及帮助函数到完善的浏览器部件。在雅虎有一支专门的软件工程师团队负责 YUI，他们提供了优秀的文档和支持。

- 协议：BSD 许可证
- 网站：http://yuilibrary.com

C.1.2 Prototype

它是一个提供了常见任务 API 的开源库。最初是针对 Ruby on Rails 框架中的使用而开发的，Prototype 是类驱动的，旨在为 JavaScript 提供类定义和继承。因此，Prototype 提供了很多类，用于将常见或复杂功能封装为简单的 API 调用。Prototype 只有一个单独的文件，可以很容易地放入任意页面。它是由 Sam Stephenson 撰写并维护的。

- 协议：MIT 许可证或者是 Creative Commons Attribution-Share Alike 3.0 Unported
- 网站：http://www.prototypejs.org/

C.1.3 Dojo Toolkit

Dojo Toolkit 开源库基于一种包系统建模，一组功能组成一个包，可以按需载入。Dojo 提供了范围广泛的选项和配置，几乎涵盖了你要用 JavaScript 做的任何事情。Dojo Toolkit 由 Alex Russell 创建，并

由 Dojo 基金会的雇员和志愿者维护。

- 协议：新 BSD 许可证或学术自由协议 2.1 版
- 网站：http://www.dojotoolkit.org/

C.1.4 MooTools

MooTools 是一个为了精简和优化而设计的开源库，它为内置 JavaScript 对象添加了各种方法，以通过接近的接口提供新功能，或者直接提供新的对象。MooTools 的短小精悍受到一些 Web 开发者的青睐。

- 协议：MIT 许可证
- 网站：http://www.mootools.net/

C.1.5 jQuery

jQuery 是一个给 JavaScript 提供了函数式编程接口的开源库。它是一个完整的库，其核心是构建于 CSS 选择器上的，用来操作 DOM 元素。通过链式调用，jQuery 代码看上去更像是对于应该发生什么的描述而不是 JavaScript 代码。这种代码风格在设计师和原型制作人中非常流行。jQuery 是由 John Resig 撰写并维护的。

- 协议：MIT 许可证或通用公共许可证（GPL）
- 网站：http://jquery.com/

C.1.6 MochiKit

MochKit 是一个由一些小工具组成的开源库，它以完善的文档和完整的测试见长，拥有大量 API 及相关范例文档以及数百个测试来确保质量。MochiKit 是由 Bob Ippolito 撰写并维护的。

- 协议：MIT 许可者或学术自由许可证 2.1 版
- 网站：http://www.mochikit.com/

C.1.7 Underscore.js

虽然严格来讲 Underscore.js 并不是一个通用的库，但它的确为 JavaScript 中的功能性编程提供了很多额外的功能。其文档称 Underscore.js 是对 jQuery 的补充，提供了操作对象、数组、函数和其他 JavaScript 数据类型的更多的低级功能。Underscore.js 由 DcoumentCloud 的 Jeremy Ashkenas 维护。

- 协议：MIT 许可证
- 网站：http://documentcloud.github.com/underscore/

C.2 互联网应用

互联网应用库是针对于简化完整的 Web 应用开发而设计的。它们并不提供应用问题的小块组件，而是提供了快速应用开发的整个概念框架。虽然这些库也可能提供一些底层功能，但他们的目标是帮助用户快速开发 Web 应用。

C.2.1 Backbone.js

Backbone.js 是构建于 Underscore.js 基础之上的一个迷你 MVC 开源库，它针对单页应用进行优化，

让你能够随着应用状态变化方便地更新页面的任意部分。Backbone.js 由 DcoumentCloud 的 Jeremy Ashkenas 维护。

- 协议：MIT 许可证
- 网站：http://documentcloud.github.com/backbone/

C.2.2 Rico

Rico 是一个开源库，旨在让行为丰富的互联网应用的开发更加简单。它提供了 Ajax、动画、样式以及部件的工具。这个库由一些志愿者组成的小团队维护，但是 2008 年起开发速度大大减慢了。

- 协议：Apache 许可证 2.0
- 网站：http://openrico.org/

C.2.3 qooxdoo

它是一个旨在协助整个互联网应用开发周期的开源库。qooxdoo 实现了它自己的类和接口，用于创建类似于传统面向对象语言的编程模型。这个库包含了一个完整的 GUI 工具包以及用于简化前端构建过程的编译器。qooxdoo 起初是 1&1webhosting 公司（www.1and1.com）的内部使用库，后来基于开源协议发布了。1&1 骋用了一些全职开发者来维护和开发这个库。

- 协议：GNU 较宽松公共许可证（LGPL）或者 Eclipse 公共许可证（EPL）
- 网站：http://www.qooxdoo.org/

C.3 动画和特效

动画和其他视觉特效也成为了 Web 开发的重要部分。在网页上做出流畅的动画是一个很重要的任务，一些开发者已经做出了易用的库，提供流畅的动画和特效。前面提到的很多通用 JavaScript 库也有动画功能。

C.3.1 script.aculo.us

script.aculo.us 是 Prototype 的“同伴”，它提供了出色特效的简单使用方式，使用的东西不超过是 CSS 和 DOM。Prototype 必须在使用 script.aculo.us 之前载入。script.aculo.us 是最流行的特效库之一，世界上很多网站和 Web 应用都在使用它。它的作者 Thomas Fuchs 积极地维护着 script.aculo.us。

- 协议：MIT 许可证
- 网站：http://script.aculo.us/

C.3.2 moo.fx

moo.fx 开源动画库是设计在 Prototype 或者 MooTools 之上运行的。它的目标是尽可能小（最新的版本是 3KB），并支持开发人员用尽可能少的代码创建动画。MooTools 是默认包含 moo.fx 的，但也可以单独下载用于 Prototype 中。

- 协议：MIT 许可证
- 网站：http://moofx.mad4milk.net/

C.3.3 Lightbox

Lightbox 是一个用于在任何页面上创建图像浮动层的 JavaScript 库，依赖于 Prototype 和 script.aculo.us 来实现它的视觉特效。基本的理念是让用户在一个浮动层中浏览一个或者一系列图像，而不必离开当前页面。Lightbox 浮动层无论是外观还是过渡效果都可以自定义。Lightbox 由 Lokesh Dhakar 开发并维护。

- **协议**：创作共用协议 2.5
- **网站**：http://www.huddletotegher.com/projects/lightbox2/

C.4 加密

随着 Ajax 应用的流行，对于浏览器端加密以确保通讯安全的需求也越来越多。幸好，一些人已经在 JavaScript 中实现了常用的安全算法。这些库大部分并没有其作者的正式支持，但还是被广泛应用着。

C.4.1 JavaScript MD5

该开源库实现了 MD4、MD5 以及 SHA-1 安全散列函数。作者 Paul Johnston 和其他一些贡献者每个算法一个文件的创建了这么丰富的库，它可以用于 Web 应用。主页上提供了散列算法的概述，对于其弱点的讨论以及适当的使用方法。

- **协议**：BSD 许可证
- **网站**：http://pajhome.org.uk/crypt/md5

C.4.2 JavaScrypt

该 JavaScript 库实现了 MD5 和 AES（256 位）加密算法。JavaScrypt 的网站提供了很多关于密码学历史及其在计算机中应用的信息。但是缺乏关于如何将该库集成到 Web 应用中的基本文档，JavaScrypt 的代码里面全都是深奥的数学处理和计算。

- **协议**：公共域
- **网站**：http://www.fourmilab.ch/javascrypt/

附录 D

JavaScript 工具

写JavaScript 代码和用其他语言编写代码很像，使用工具能够提高工作效率。JavaScript 开发人员可用的工具数量一度爆发性增长，使得查找问题、优化和部署基于 JavaScript 的解决方案更为简单。其中一些工具是专为 JavaScript 设计使用的，而其他一些可以在浏览器之外运行。本附录对其中一些工具给出了概述，并额外提供了信息资源。

D.1 校验器

JavaScript 调试有一个问题，很多 IDE 并不能在输入的时候自动指出语法错误。大多数开发者写了一部分代码之后要将其载入到浏览器中查找错误。你可以通过在部署之前校验 JavaScript 代码，以便显著地减少此类错误。校验器提供了基本的语法检查，并给出某些风格的警告。

D.1.1 JSLint

JSLint 是一个由 Douglas Crockford 撰写的 JavaScript 校验器。它可以从核心层次上检查语法错误，伴随跨浏览器问题的最小共同点检查（它遵循最严格的规则来确保代码到处都能运行）。你可以启用 Crockford 对于代码风格的警告，包括代码格式、未声明的全局变量的使用以及其他更多警告。尽管 JSLint 是用 JavaScript 写的，但是通过基于 Java 的 Rhino 解释器，它可以在命令行中运行，或者通过 WScript 或者其他 JavaScript 解释器。网站上提供了针对各种命令行解释器的自定义版本。

- 价格：免费
- 网站：http://www.jslint.com/

D.1.2 JSHint

JSHint 是 JSLint 的一个分支，为应用规则提供了更多的自定义功能。与 JSLint 类似，它首先检查语法错误，然后检查有问题的编码模式。JSLint 的每一项检查 JSHint 都有，但开发人员可以更好地控制应用什么规则。与 JSLint 一样，JSHint 也能使用 Rhino 在命令行中运行。

- 价格：免费
- 网站：http://www.jshint.com/

D.1.3 JavaScript Lint

它和 JSLint 完全不相干，JavaScript Lint 是 Matthias Miller 写的一个基于 C 的 JavaScript 校验器。它使用了 SpiderMonkey（即 Firefox 所用的 JavaScript 解释器）来分析代码并查找语法错误。这个工具包含

大量选项，可以启用额外关于编码风格的警告，以及未声明的变量和不可到达的代码警告。Windows 和 Macintosh 上都有可用的 JavaScript Lint，源代码也可以自由取得。

- 价格：免费
- 网站：http://www.javascriptlint.com/

D.2 压缩器

JavaScript 构建过程中很重要的一部分，是压缩输出并移除多余的字符。这样做可以确保传送到浏览器的字节数最小化，最终加速了用户体验。有几种压缩比率不同的工具可以选择。D.2.1 JSMin

JSMin 是由 Douglas Crockford 写的一个基于 C 的压缩器，进行最基本的 JavaScript 压缩。它主要是移除空白和注释，确保最终的代码依然可以被顺利执行。JSMin 有 Windows 执行程序，包括 C 版本代码，还有其他语言的代码：

- 价格：免费
- 网站：http://www.crockford.com/javascript/jsmin.html

D.2.2 Dojo ShrinkSafe

负责 Dojo Toolkit 的同一批人开发了一个叫做 ShrinkSafe 的工具，它使用了 Rhino JavaScript 解释器首先将 JavaScript 代码解析为记号流，然后用它们来安全压缩代码。和 JSMin 一样，ShrinkSafe 移除多余的空白符（不包括换行）和注释，但是还更进一步将局部变量替换为两个字符长的变量名。最后可以比 JSMin 产生更小输出，而没有引入语法错误的风险。

- 价格：免费
- 网站：http:// shrinksafe .dojotoolkit.org/

D.2.3 YUI Compressor

YUI 小组有一个叫做 YUI Compressor 的压缩器。和 ShrinkSafe 类似，YUI Compressor 利用了 Rhino 解释器将 JavaScript 代码解析为记号流，并移除注释和空白字符并替换变量名。与 ShrinkSafe 不同，YUI Compressor 还移除换行并进行一些细微的优化进一步节省字节数。一般来说，YUI Compressor 处理过的文件要小于 JSMin 或者 ShrinkSafe 处理过的文件。

- 价格：免费
- 网站：http://yuilibrary.com /projects/yuicompressor

D.3 单元测试

TDD（Test-driven development，测试驱动开发）是一种以单元测试为核心的软件开发过程。直到最近，才出现了一些在 JavaScript 进行单元测试的工具。现在多数 JavaScript 库都在它们自己的代码中使用了某种形式的单元测试，其中一些发布了单元测试框架让他人使用。

D.3.1 JsUnit

最早的 JavaScript 单元测试框架，不绑定于任何特定的 JavaScript 库。JsUnit 是 Java 知名的 JUnit 测

试框架的移植。测试在页面中运行，并可以设置为自动测试并将结果提交到服务器。它的网站上包含了例子和基本的文档：

- **价格**：免费
- **网站**：http://www.jsunit.net/

D.3.2 YUI Test

作为 YUI 的一部分，YUI Test 不仅可以用于测试使用 YUI 的代码，也可以测试网站或者应用中的任何代码。YUI Test 包含了简单和复杂的断言，以及一种模拟简单的鼠标和键盘事件的方法。该框架在 Yahoo! Developer Network 上有完整的文档描述，包含了例子、API 文档和更多内容。测试时在浏览器中运行，结果输出在页面上。YUI 便使用 YUI Test 来测试整个库。

- **价格**：免费
- **网站**：http://yuilibrary.com /projects/yuitest/

D.3.3 DOH

DOH（Dojo Object Harness）在发布给大家使用之前，最初是作为 Dojo 内部的单元测试工具出现的。和其他框架一样，单元测试是在浏览器中运行的。

- **价格**：免费
- **网站**：http://www.dojotoolkit.org/

D.3.4 qUnit

qUnit 是为测试 jQuery 而开发的一个单元测试框架。jQuery 本身的确使用 qUnit 进行各项测试。除此之外，qUnit 与 jQuery 并没有绑定关系，也可以用它来测试所有 JavaScript 代码。qUnit 的特点是简单易用，一般开发人员很容易上手。

- **价格**：免费
- **网站**：https://github.com/jquery/qunit

D.4 文档生成器

大多数 IDE 对于主流语言都包含了文档生成器。由于 JavaScript 并没有官方的 IDE，过去文档一般都是手工完成，或者是利用针对其他语言的文档生成器。然而，现在终于有一些专门针对 JavaScript 的文档生成器了。

D.4.1 JsDoc Toolkit

JsDoc Toolkit 是最早出现的 JavaScript 文档生成器之一。它要求你在代码中输入类似 Javadoc 的注释，然后处理这些注释并输出为 HTML 文件。你可以自定义 HTML 的格式，这需使用预定义的 JsDoc 模板或者创建自己的模版。JsDoc Toolkit 可以以 Java 包的形式获得。

- **价格**：免费
- **网站**：http://code.google.com/p/jsdoc-toolkit/

D.4.2 YUI Doc

YUI Doc 是 YUI 的文档生成器。该生成器以 Python 书写，所以它要求安装有 Python 运行时环境。YUI Doc 可以输出集成了属性和方法搜索（用 YUI 的自动完成挂件实现的）的 HTML 文件。和 JsDoc 一样，YUI Doc 要求源代码中使用类似 Javadoc 的注释。默认的 HTML 可以通过修改默认的 HTML 模板文件和相关的样式表来更改。

- 价格：免费
- 网站：http:// www.yuilibrary.com/projects/yuidoc/

D.4.3 AjaxDoc

AjaxDoc 的目标和前面提到的生成器有些差异。它不为 JavaScript 文档生成 HTML 文件，而是创建与针对.NET 语言（如 C#和 Visual Basic. NET）所创建文件相同格式的 XML 文件。这样做就可以由标准的.NET 文档生成器创建 HTML 文件形式的文档。AjaxDoc 使用类似于所有.NET 语言用到的文档注释格式。创建 AjaxDoc 是针对 ASP.NET 的 Ajax 解决方案，但是它也可以用于单独的项目。

- 价格：免费
- 网站：http://www.codeplex.com/ajaxdoc

D.5 安全执行环境

随着 mashup 应用越来越流行，对于允许来自外界的 JavaScript 存在于同一个页面上并执行有着越来越多的需求。这导致了一些访问受限功能的安全问题。以下工具旨在创建安全的执行环境，其中不同来源的 JavaScript 可以共存，而不会互相影响。

D.5.1 ADsafe

由 Douglas Crockford 创建，ADsafe 是 JavaScript 的子集，这个子集被认为可以被第三方脚本安全访问。对于用 ADsafe 运行的代码，页面必须包含 ADsafe JavaScript 库并标记为 ADsafe 挂件格式。因此，代码可以在任何页面上安全执行。

- 价格：免费
- 网站：http://www.adsafe.org/

D.5.2 Caja

Caja 用一种独特的方式来确保 JavaScript 的安全执行。类似于 ADsafe，Caja 定义了 JavaScript 的一个可以用安全方式使用的子集。Caja 继而可以清理 JavaScript 代码并验证它只按照预期的方式运行。作为该项目的一部分，有一种叫做 Cajita 的语言，它是 JavaScript 功能的一种更小的子集。Caja 还处于幼年期，但是已经展示了很多前景，允许多个脚本在同一个页面执行而没有恶意活动的可能。

- 价格：免费
- 网站：http://code.google.com/p/google-caja/