

# 第 22 章

## 高级技巧

### 本章内容

- 使用高级函数
- 防篡改对象
- Yielding Timers

JavaScript 是一种极其灵活的语言，具有多种使用风格。一般来说，编写 JavaScript 要么使用过程方式，要么使用面向对象方式。然而，由于它天生的动态属性，这种语言还能使用更为复杂和有趣的模式。这些技巧要利用 ECMAScript 的语言特点、BOM 扩展和 DOM 功能来获得强大的效果。

### 22.1 高级函数

函数是 JavaScript 中最有趣的部分之一。它们本质上是十分简单和过程化的，但也可以是非常复杂和动态的。一些额外的功能可以通过使用闭包来实现。此外，由于所有的函数都是对象，所以使用函数指针非常简单。这些令 JavaScript 函数不仅有趣而且强大。以下几节描绘了几种在 JavaScript 中使用函数的高级方法。

#### 22.1.1 安全的类型检测

JavaScript 内置的类型检测机制并非完全可靠。事实上，发生错误否定及错误肯定的情况也不在少数。比如说 `typeof` 操作符吧，由于它有一些无法预知的行为，经常会导致检测数据类型时得到不靠谱的结果。Safari（直至第 4 版）在对正则表达式应用 `typeof` 操作符时会返回 `"function"`，因此很难确定某个值到底是不是函数。

再比如，`instanceof` 操作符在存在多个全局作用域（像一个页面包含多个框架）的情况下，也是问题多多。一个经典的例子（第 5 章也提到过）就是像下面这样将对象标识为数组。

```
var isArray = value instanceof Array;
```

以上代码要返回 `true`，`value` 必须是一个数组，而且还必须与 `Array` 构造函数在同个全局作用域中。（别忘了，`Array` 是 `window` 的属性。）如果 `value` 是在另一个框架中定义的数组，那么以上代码就会返回 `false`。

在检测某个对象到底是原生对象还是开发人员自定义的对象的时候，也会有问题。出现这个问题的原因是浏览器开始原生支持 JSON 对象了。因为很多人一直在使用 Douglas Crockford 的 JSON 库，而该库定义了一个全局 JSON 对象。于是开发人员很难确定页面中的 JSON 对象到底是不是原生的。

解决上述问题的办法都一样。大家知道，在任何值上调用 `Object` 原生的 `toString()` 方法，都会

返回一个[object NativeConstructorName]格式的字符串。每个类在内部都有一个[[Class]]属性，这个属性中就指定了上述字符串中的构造函数名。举个例子吧。

```
alert(Object.prototype.toString.call(value));    //[object Array]"
```

由于原生数组的构造函数名与全局作用域无关，因此使用 toString() 就能保证返回一致的值。利用这一点，可以创建如下函数：


```
function isArray(value){
    return Object.prototype.toString.call(value) == "[object Array]";
}
```

同样，也可以基于这一思路来测试某个值是不是原生函数或正则表达式：

```
function isFunction(value){
    return Object.prototype.toString.call(value) == "[object Function]";
}
function isRegExp(value){
    return Object.prototype.toString.call(value) == "[object RegExp]";
}
```

不过要注意，对于在 IE 中以 COM 对象形式实现的任何函数，isFunction() 都将返回 false（因为它们并非原生的 JavaScript 函数，请参考第 10 章中更详细的介绍）。

这一技巧也广泛应用于检测原生 JSON 对象。Object 的 toString() 方法不能检测非原生构造函数的构造函数名。因此，开发人员定义的任何构造函数都将返回[object Object]。有些 JavaScript 库会包含与下面类似的代码。



```
var isNativeJSON = window.JSON && Object.prototype.toString.call(JSON) ==
    "[object JSON]";
```


在 Web 开发中能够区分原生与非原生 JavaScript 对象非常重要。只有这样才能确切知道某个对象到底有哪些功能。这个技巧可以对任何对象给出正确的结论。



请注意，Object.prototype.toString() 本身也可能被修改。本节讨论的技巧假设 Object.prototype.toString() 是未被修改过的原生版本。

## 22.1.2 作用域安全的构造函数

第 6 章讲述了用于自定义对象的构造函数的定义和用法。你应该还记得，构造函数其实就是一个使用 new 操作符调用的函数。当使用 new 调用时，构造函数内用到的 this 对象会指向新创建的对象实例，如下面的例子所示：



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

```
var person = new Person("Nicholas", 29, "Software Engineer");
```

上面这个例子中，Person 构造函数使用 this 对象给三个属性赋值：name、age 和 job。当和 new 操作符连用时，则会创建一个新的 Person 对象，同时会给它分配这些属性。问题出在当没有使用 new 操作符来调用该构造函数的情况上。由于该 this 对象是在运行时绑定的，所以直接调用 Person()，this 会映射到全局对象 window 上，导致错误对象属性的意外增加。例如：


```
var person = Person("Nicholas", 29, "Software Engineer");
alert(window.name);           //"Nicholas"
alert(window.age);            //29
alert(window.job);            //"Software Engineer"
```

---

#### *ScopeSafeConstructorsExample01.htm*

这里，原本针对 Person 实例的三个属性被加到 window 对象上，因为构造函数是作为普通函数调用的，忽略了 new 操作符。这个问题是由 this 对象的晚绑定造成的，在这里 this 被解析成了 window 对象。由于 window 的 name 属性是用于识别链接目标和框架的，所以这里对该属性的偶然覆盖可能会导致该页面上出现其他错误。这个问题的解决方法就是创建一个作用域安全的构造函数。

作用域安全的构造函数在进行任何更改前，首先确认 this 对象是正确类型的实例。如果不是，那么会创建新的实例并返回。请看以下例子：



```
function Person(name, age, job){
    if (this instanceof Person){
        this.name = name;
        this.age = age;
        this.job = job;
    } else {
        return new Person(name, age, job);
    }
}

var person1 = Person("Nicholas", 29, "Software Engineer");
alert(window.name);           //" "
alert(person1.name);          //"Nicholas"

var person2 = new Person("Shelby", 34, "Ergonomist");
alert(person2.name);          //"Shelby"
```

---

#### *ScopeSafeConstructorsExample02.htm*

这段代码中的 Person 构造函数添加了一个检查并确保 this 对象是 Person 实例的 if 语句，它表示要么使用 new 操作符，要么在现有的 Person 实例环境中调用构造函数。任何一种情况下，对象初始化都能正常进行。如果 this 并非 Person 的实例，那么会再次使用 new 操作符调用构造函数并返回结果。最后的结果是，调用 Person 构造函数时无论是否使用 new 操作符，都会返回一个 Person 的新实例，这就避免了在全局对象上意外设置属性。

关于作用域安全的构造函数的贴心提示。实现这个模式后，你就锁定了可以调用构造函数的环境。如果你使用构造函数窃取模式的继承且不使用原型链，那么这个继承很可能被破坏。这里有个例子：

```
function Polygon(sides){
    if (this instanceof Polygon) {
        this.sides = sides;
        this.getArea = function(){
```

```

        return 0;
    };
} else {
    return new Polygon(sides);
}
}

function Rectangle(width, height){
    Polygon.call(this, 2);
    this.width = width;
    this.height = height;
    this.getArea = function(){
        return this.width * this.height;
    };
}


var rect = new Rectangle(5, 10);
alert(rect.sides);           //undefined

```

### ScopeSafeConstructorsExample03.htm

在这段代码中，Polygon 构造函数是作用域安全的，然而 Rectangle 构造函数则不是。新创建一个 Rectangle 实例之后，这个实例应该通过 Polygon.call() 来继承 Polygon 的 sides 属性。但是，由于 Polygon 构造函数是作用域安全的，this 对象并非 Polygon 的实例，所以会创建并返回一个新的 Polygon 对象。Rectangle 构造函数中的 this 对象并没有得到增长，同时 Polygon.call() 返回的值也没有用到，所以 Rectangle 实例中就不会有 sides 属性。

如果构造函数窃取结合使用原型链或者寄生组合则可以解决这个问题。考虑以下例子：



```

function Polygon(sides){
    if (this instanceof Polygon) {
        this.sides = sides;
        this.getArea = function(){
            return 0;
        };
    } else {
        return new Polygon(sides);
    }
}

function Rectangle(width, height){
    Polygon.call(this, 2);
    this.width = width;
    this.height = height;
    this.getArea = function(){
        return this.width * this.height;
    };
}

Rectangle.prototype = new Polygon();

var rect = new Rectangle(5, 10);
alert(rect.sides);           //2

```

### ScopeSafeConstructorsExample04.htm

上面这段重写的代码中,一个 Rectangle 实例也同时是一个 Polygon 实例,所以 Polygon.call() 会照原意执行,最终为 Rectangle 实例添加了 sides 属性。

多个程序员在同一个页面上写 JavaScript 代码的环境中,作用域安全构造函数就很有用了。届时,对全局对象意外的更改可能会导致一些常常难以追踪的错误。除非你单纯基于构造函数窃取来实现继承,推荐作用域安全的构造函数作为最佳实践。

### 22.1.3 惰性载入函数

因为浏览器之间行为的差异,多数 JavaScript 代码包含了大量的 if 语句,将执行引导到正确的代码中。看看下面来自上一章的 createXHR() 函数。

```
function createXHR(){
    if (typeof XMLHttpRequest != "undefined"){
        return new XMLHttpRequest();
    } else if (typeof ActiveXObject != "undefined"){
        if (typeof arguments.callee.activeXString != "string"){
            var versions = ["MSXML2.XMLHttp.6.0", "MSXML2.XMLHttp.3.0",
                           "MSXML2.XMLHttp"],
                i, len;

            for (i=0, len=versions.length; i < len; i++){
                try {
                    new ActiveXObject(versions[i]);
                    arguments.callee.activeXString = versions[i];
                    break;
                } catch (ex){
                    //跳过
                }
            }

            return new ActiveXObject(arguments.callee.activeXString);
        } else {
            throw new Error("No XHR object available.");
        }
    }
}
```

每次调用 createXHR() 的时候,它都要对浏览器所支持的能力仔细检查。首先检查内置的 XHR,然后测试有没有基于 ActiveX 的 XHR,最后如果都没有发现的话就抛出一个错误。每次调用该函数都是这样,即使每次调用时分支的结果都不变:如果浏览器支持内置 XHR,那么它就一直支持了,那么这种测试就变得没必要了。即使只有一个 if 语句的代码,也肯定要比没有 if 语句的慢,所以如果 if 语句不必每次执行,那么代码可以运行地更快一些。解决方案就是称之为惰性载入的技巧。

惰性载入表示函数执行的分支仅会发生一次。有两种实现惰性载入的方式,第一种就是在函数被调用时再处理函数。在第一次调用的过程中,该函数会被覆盖为另外一个按合适方式执行的函数,这样任何对原函数的调用都不用再经过执行的分支了。例如,可以用下面的方式使用惰性载入重写 createXHR()。

```
function createXHR(){
    if (typeof XMLHttpRequest != "undefined"){
        createXHR = function(){
```



```

        return new XMLHttpRequest();
    };
    } else if (typeof ActiveXObject != "undefined"){
        createXHR = function(){
            if (typeof arguments.callee.activeXString != "string"){
                var versions = ["MSXML2.XMLHttp.6.0", "MSXML2.XMLHttp.3.0",
                               "MSXML2.XMLHttp"],
                    i, len;

                for (i=0,len=versions.length; i < len; i++){
                    try {
                        new ActiveXObject(versions[i]);
                        arguments.callee.activeXString = versions[i];
                        break;
                    } catch (ex){
                        //skip
                    }
                }

                return new ActiveXObject(arguments.callee.activeXString);
            };
        } else {
            createXHR = function(){
                throw new Error("No XHR object available.");
            };
        }


        return createXHR();
    }
}

```

*LazyLoadingExample01.htm*

在这个惰性载入的 createXHR() 中, if 语句的每一个分支都会为 createXHR 变量赋值, 有效覆盖了原有的函数。最后一步便是调用新赋的函数。下一次调用 createXHR() 的时候, 就会直接调用被分配的函数, 这样就不用再次执行 if 语句了。

第二种实现惰性载入的方式是在声明函数时就指定适当的函数。这样, 第一次调用函数时就不会损失性能了, 而在代码首次加载时会损失一点性能。以下就是按照这一思路重写前面例子的结果。



```

var createXHR = (function(){
    if (typeof XMLHttpRequest != "undefined"){
        return function(){
            return new XMLHttpRequest();
        };
    } else if (typeof ActiveXObject != "undefined"){
        return function(){
            if (typeof arguments.callee.activeXString != "string"){
                var versions = ["MSXML2.XMLHttp.6.0", "MSXML2.XMLHttp.3.0",
                               "MSXML2.XMLHttp"],
                    i, len;

                for (i=0,len=versions.length; i < len; i++){
                    try {
                        new ActiveXObject(versions[i]);
                        arguments.callee.activeXString = versions[i];
                        break;
                    } catch (ex){
                        //skip
                    }
                }

                return new ActiveXObject(arguments.callee.activeXString);
            };
        } else {
            return function(){
                throw new Error("No XHR object available.");
            };
        }
    }
})();

```

```

    }
    return new ActiveXObject(arguments.callee.activeXString);
};
} else {
    return function(){
        throw new Error("No XHR object available.");
    };
}
})();

```

[LazyLoadingExample02.htm](#)

这个例子中使用的技巧是创建一个匿名、自执行的函数，用以确定应该使用哪一个函数实现。实际的逻辑都一样。不一样的地方就是第一行代码（使用 var 定义函数）、新增了自执行的匿名函数，另外每个分支都返回正确的函数定义，以便立即将其赋值给 createXHR()。

惰性载入函数的优点是只在执行分支代码时牺牲一点儿性能。至于哪种方式更合适，就要看你的具体需求而定了。不过这两种方式都能避免执行不必要的代码。

## 22.1.4 函数绑定

另一个日益流行的高级技巧叫做函数绑定。函数绑定要创建一个函数，可以在特定的 this 环境中以指定参数调用另一个函数。该技巧常常和回调函数与事件处理程序一起使用，以便在将函数作为变量传递的同时保留代码执行环境。请看以下例子：

```

var handler = {
    message: "Event handled",

    handleClick: function(event){
        alert(this.message);
    }
};

var btn = document.getElementById("my-btn");
EventUtil.addHandler(btn, "click", handler.handleClick);

```

在上面这个例子中，创建了一个叫做 handler 的对象。handler.handleClick() 方法被分配为一个 DOM 按钮的事件处理程序。当按下该按钮时，就调用该函数，显示一个警告框。虽然貌似警告框应该显示 Event handled，然而实际上显示的是 undefiend。这个问题在于没有保存 handler.handleClick() 的环境，所以 this 对象最后是指向了 DOM 按钮而非 handler（在 IE8 中，this 指向 window。）可以如下面例子所示，使用一个闭包来修正这个问题。

```

var handler = {
    message: "Event handled",


    handleClick: function(event){
        alert(this.message);
    }
};

var btn = document.getElementById("my-btn");
EventUtil.addHandler(btn, "click", function(event){
    handler.handleClick(event);
});

```

这个解决方案在 onclick 事件处理程序内使用了一个闭包直接调用 handler.handleClick()。当然,这是特定于这段代码的解决方案。创建多个闭包可能会令代码变得难于理解和调试。因此,很多 JavaScript 库实现了一个可以将函数绑定到指定环境的函数。这个函数一般都叫 bind()。

一个简单的 bind() 函数接受一个函数和一个环境,并返回一个在给定环境中调用给定函数的函数,并且将所有参数原封不动传递过去。语法如下:



```
function bind(fn, context){
    return function(){
        return fn.apply(context, arguments);
    };
}
```

---

*FunctionBindingExample01.htm*

这个函数似乎简单,但其功能是非常强大的。在 bind() 中创建了一个闭包,闭包使用 apply() 调用传入的函数,并给 apply() 传递 context 对象和参数。注意这里使用的 arguments 对象是内部函数的,而非 bind() 的。当调用返回的函数时,它会在给定环境中执行被传入的函数并给出所有参数。bind() 函数按如下方式使用:

```
var handler = {
    message: "Event handled",


    handleClick: function(event){
        alert(this.message);
    }
};

var btn = document.getElementById("my-btn");
EventUtil.addHandler(btn, "click", bind(handler.handleClick, handler));
```

---

*FunctionBindingExample01.htm*

在这个例子中,我们用 bind() 函数创建了一个保持了执行环境的函数,并将其传给 EventUtil.addHandler()。event 对象也被传给了该函数,如下所示:



```
var handler = {
    message: "Event handled",

    handleClick: function(event){
        alert(this.message + ":" + event.type);
    }
};

var btn = document.getElementById("my-btn");
EventUtil.addHandler(btn, "click", bind(handler.handleClick, handler));
```

---

*FunctionBindingExample01.htm*

handler.handleClick() 方法和平时一样获得了 event 对象,因为所有的参数都通过被绑定的函数直接传给了它。

ECMAScript 5 为所有函数定义了一个原生的 bind() 方法,进一步简单了操作。换句话说,你不用再自己定义 bind() 函数了,而是可以直接在函数上调用这个方法。例如:



```

var handler = {
    message: "Event handled",

    handleClick: function(event){
        alert(this.message + ":" + event.type);
    }
};

var btn = document.getElementById("my-btn");
EventUtil.addHandler(btn, "click", handler.handleClick.bind(handler));

```

*FunctionBindingExample02.htm*

原生的 `bind()` 方法与前面介绍的自定义 `bind()` 方法类似，都是要传入作为 `this` 值的对象。支持原生 `bind()` 方法的浏览器有 IE9+、Firefox 4+ 和 Chrome。

只要是将某个函数指针以值的形式进行传递，同时该函数必须在特定环境中执行，被绑定函数的效用就突显出来了。它们主要用于事件处理程序以及 `setTimeout()` 和 `setInterval()`。然而，被绑定函数与普通函数相比有更多的开销，它们需要更多内存，同时也因为多重函数调用稍微慢一点，所以最好只在必要时使用。

## 22.1.5 函数柯里化

与函数绑定紧密相关的主题是函数柯里化 (function currying)，它用于创建已经设置好了一个或多个参数的函数。函数柯里化的基本方法和函数绑定是一样的：使用一个闭包返回一个函数。两者的区别在于，当函数被调用时，返回的函数还需要设置一些传入的参数。请看以下例子。

```

function add(num1, num2){
    return num1 + num2;
}

function curriedAdd(num2){
    return add(5, num2);
}

alert(add(2, 3)); //5
alert(curriedAdd(3)); //8

```

这段代码定义了两个函数：`add()` 和 `curriedAdd()`。后者本质上是在任何情况下第一个参数为 5 的 `add()` 版本。尽管从技术上来说 `curriedAdd()` 并非柯里化的函数，但它很好地展示了其概念。

柯里化函数通常由以下步骤动态创建：调用另一个函数并为它传入要柯里化的函数和必要参数。下面是创建柯里化函数的通用方式。


```

function curry(fn){
    var args = Array.prototype.slice.call(arguments, 1);
    return function(){
        var innerArgs = Array.prototype.slice.call(arguments);
        var finalArgs = args.concat(innerArgs);
        return fn.apply(null, finalArgs);
    };
}

```

*FunctionCurryingExample01.htm*

`curry()` 函数的主要工作就是将被返回函数的参数进行排序。`curry()` 的第一个参数是要进行柯里化的函数，其他参数是要传入的值。为了获取第一个参数之后的所有参数，在 `arguments` 对象上调用了 `slice()` 方法，并传入参数 1 表示被返回的数组包含从第二个参数开始的所有参数。然后 `args` 数组包含了来自外部函数的参数。在内部函数中，创建了 `innerArgs` 数组用来存放所有传入的参数（又一次用到了 `slice()`）。有了存放来自外部函数和内部函数的参数数组后，就可以使用 `concat()` 方法将它们组合为 `finalArgs`，然后使用 `apply()` 将结果传递给该函数。注意这个函数并没有考虑到执行环境，所以调用 `apply()` 时第一个参数是 `null`。`curry()` 函数可以按以下方式应用。



```
function add(num1, num2){
    return num1 + num2;
}

var curriedAdd = curry(add, 5);
alert(curriedAdd(3)); //8
```

---

*FunctionCurryingExample01.htm*

在这个例子中，创建了第一个参数绑定为 5 的 `add()` 的柯里化版本。当调用 `curriedAdd()` 并传入 3 时，3 会成为 `add()` 的第二个参数，同时第一个参数依然是 5，最后结果便是和 8。你也可以像下面例子这样给出所有的函数参数：

```
function add(num1, num2){
    return num1 + num2;
}

var curriedAdd = curry(add, 5, 12);
alert(curriedAdd()); //17
```

---

*FunctionCurryingExample01.htm*

在这里，柯里化的 `add()` 函数两个参数都提供了，所以以后就无需再传递它们了。函数柯里化还常常作为函数绑定的一部分包含在其中，构造出更为复杂的 `bind()` 函数。例如：

```
function bind(fn, context){
    var args = Array.prototype.slice.call(arguments, 2);
    return function(){
        var innerArgs = Array.prototype.slice.call(arguments);
        var finalArgs = args.concat(innerArgs);
        return fn.apply(context, finalArgs);
    };
}
```

---

*FunctionCurryingExample02.htm*

对 `curry()` 函数的主要更改在于传入的参数个数，以及它如何影响代码的结果。`curry()` 仅仅接受一个要包裹的函数作为参数，而 `bind()` 同时接受函数和一个 `object` 对象。这表示给被绑定的函数的参数是从第三个开始而不是第二个，这就要更改 `slice()` 的第一处调用。另一处更改是在倒数第 3 行将 `object` 对象传给 `apply()`。当使用 `bind()` 时，它会返回绑定到给定环境的函数，并且可能它其中某些函数参数已经被设好。当你想除了 `event` 对象再额外给事件处理程序传递参数时，这非常有用，例如：



```
var handler = {
  message: "Event handled",

  handleClick: function(name, event){
    alert(this.message + ":" + name + ":" + event.type);
  }
};

var btn = document.getElementById("my-btn");
EventUtil.addHandler(btn, "click", bind(handler.handleClick, handler, "my-btn"));
```

---

*FunctionCurryingExample02.htm*

在这个更新过的例子中，handler.handleClick()方法接受了两个参数：要处理的元素的名字和 event 对象。作为第三个参数传递给 bind() 函数的名字，又被传递给了 handler.handleClick()，而 handler.handleClick() 也会同时接收到 event 对象。

ECMAScript 5 的 bind() 方法也实现函数柯里化，只要在 this 的值之后再传入另一个参数即可。

```
var handler = {
  message: "Event handled",

  handleClick: function(name, event){
    alert(this.message + ":" + name + ":" + event.type);
  }
};

var btn = document.getElementById("my-btn");
EventUtil.addHandler(btn, "click", handler.handleClick.bind(handler, "my-btn"));
```

---

*FunctionCurryingExample03.htm*

JavaScript 中的柯里化函数和绑定函数提供了强大的动态函数创建功能。使用 bind() 还是 curry() 要根据是否需要 object 对象响应来决定。它们都能用于创建复杂的算法和功能，当然两者都不应滥用，因为每个函数都会带来额外的开销。

## 22.2 防篡改对象

JavaScript 共享的本质一直是开发人员心头的痛。因为任何对象都可以被在同一环境中运行的代码修改。开发人员很可能会意外地修改别人的代码，甚至更糟糕地，用不兼容的功能重写原生对象。ECMAScript 5 致力于解决这个问题，可以让开发人员定义防篡改对象 (tamper-proof object)。

第6章讨论了对象属性的问题，也讨论了如何手工设置每个属性的 [[Configurable]]、[[Writable]]、[[Enumerable]]、[[Value]]、[[Get]] 以及 [[Set]] 特性，以改变属性的行为。类似地，ECMAScript 5 也增加了几个方法，通过它们可以指定对象的行为。

不过请注意：一旦把对象定义为防篡改，就无法撤销了。

### 22.2.1 不可扩展对象

默认情况下，所有对象都是可以扩展的。也就是说，任何时候都可以向对象中添加属性和方法。例如，可以像下面这样先定义一个对象，后来再给它添加一个属性。

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

即使第一行代码已经完整定义 `person` 对象，但第二行代码仍然能给它添加属性。现在，使用 `Object.preventExtensions()` 方法可以改变这个行为，让你不能再给对象添加属性和方法。例如：

```
var person = { name: "Nicholas" };
Object.preventExtensions(person);


person.age = 29;
alert(person.age); //undefined
```

---

*NonExtensibleObjectsExample01.htm*

在调用了 `Object.preventExtensions()` 方法后，就不能给 `person` 对象添加新属性和方法了。在非严格模式下，给对象添加新成员会导致静默失败，因此 `person.age` 将是 `undefined`。而在严格模式下，尝试给不可扩展的对象添加新成员会导致抛出错误。

虽然不能给对象添加新成员，但已有的成员则丝毫不受影响。你仍然还可以修改和删除已有的成员。另外，使用 `Object.isExtensible()` 方法还可以确定对象是否可以扩展。



```
var person = { name: "Nicholas" };
alert(Object.isExtensible(person));    //true

Object.preventExtensions(person);
alert(Object.isExtensible(person));    //false
```


---

*NonExtensibleObjectsExample02.htm*

## 22.2.2 密封的对象

ECMAScript 5 为对象定义的第二个保护级别是密封对象（sealed object）。密封对象不可扩展，而且已有成员的 `[[Configurable]]` 特性将被设置为 `false`。这就意味着不能删除属性和方法，因为不能使用 `Object.defineProperty()` 把数据属性修改为访问器属性，或者相反。属性值是可以修改的。

要密封对象，可以使用 `Object.seal()` 方法。



```
var person = { name: "Nicholas" };
Object.seal(person);

person.age = 29;
alert(person.age);    //undefined

delete person.name;
alert(person.name);    //"Nicholas"
```

---

*SealedObjectsExample01.htm*

在这个例子中，添加 `age` 属性的行为被忽略了。而尝试删除 `name` 属性的操作也被忽略了，因此这个属性没有受任何影响。这是非严格模式下的行为。在严格模式下，尝试添加或删除对象成员都会导致抛出错误。

使用 `Object.isSealed()` 方法可以确定对象是否被密封了。因为被密封的对象不可扩展，所以用 `Object.isExtensible()` 检测密封的对象也会返回 `false`。

```
var person = { name: "Nicholas" };
alert(Object.isExtensible(person)); //true
alert(Object.isSealed(person));    //false

Object.seal(person);
alert(Object.isExtensible(person)); //false
alert(Object.isSealed(person));    //true
```

---

*SealedObjectsExample02.htm*

## 22.2.3 冻结的对象

最严格的防篡改级别是冻结对象（frozen object）。冻结的对象既不可扩展，又是密封的，而且对象数据属性的[[Writable]]特性会被设置为 `false`。如果定义[[Set]]函数，访问器属性仍然是可写的。ECMAScript 5 定义的 `Object.freeze()` 方法可以用来冻结对象。

```
var person = { name: "Nicholas" };
Object.freeze(person);

person.age = 29;
alert(person.age);    //undefined

delete person.name;
alert(person.name);   //"Nicholas"


person.name = "Greg";
alert(person.name);   //"Nicholas"
```

---

*FrozenObjectsExample01.htm*

与密封和不允许扩展一样，对冻结的对象执行非法操作在非严格模式下会被忽略，而在严格模式下会抛出错误。

当然，也有一个 `Object.isFrozen()` 方法用于检测冻结对象。因为冻结对象既是密封的又是不可扩展的，所以用 `Object.isExtensible()` 和 `Object.isSealed()` 检测冻结对象将分别返回 `false` 和 `true`。



```
var person = { name: "Nicholas" };
alert(Object.isExtensible(person)); //true
alert(Object.isSealed(person));    //false
alert(Object.isFrozen(person));    //false

Object.freeze(person);
alert(Object.isExtensible(person)); //false
alert(Object.isSealed(person));    //true
alert(Object.isFrozen(person));    //true
```

---

*FrozenObjectsExample02.htm*

对 JavaScript 库的作者而言,冻结对象是很有用的。因为 JavaScript 库最怕有人意外(或有意)地修改了库中的核心对象。冻结(或密封)主要的库对象能够防止这些问题的发生。

## 22.3 高级定时器

使用 `setTimeout()` 和 `setInterval()` 创建的定时器可以用于实现有趣且有用的功能。虽然人们对 JavaScript 的定时器存在普遍的误解,认为它们是线程,其实 JavaScript 是运行于单线程的环境中的,而定时器仅仅只是计划代码在未来的某个时间执行。执行时机是不能保证的,因为在页面的生命周期中,不同时间可能有其他代码在控制 JavaScript 进程。在页面下载完后的代码运行、事件处理程序、Ajax 回调函数都必须使用同样的线程来执行。实际上,浏览器负责进行排序,指派某段代码在某个时间点运行的优先级。

可以把 JavaScript 想象成在时间线上运行的。当页面载入时,首先执行是任何包含在 `<script>` 元素中的代码,通常是页面生命周期后面要用到的一些简单的函数和变量的声明,不过有时候也包含一些初始数据的处理。在这之后,JavaScript 进程将等待更多代码执行。当进程空闲的时候,下一个代码会被触发并立刻执行。例如,当点击某个按钮时, `onclick` 事件处理程序会立刻执行,只要 JavaScript 进程处于空闲状态。这样一个页面的时间线类似于图 22-1。

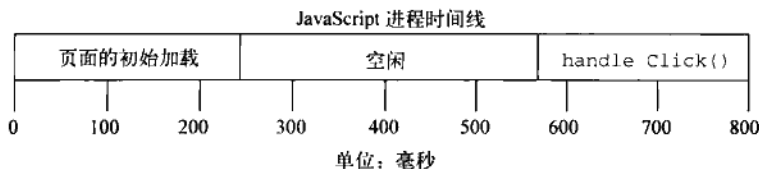


图 22-1

除了主 JavaScript 执行进程外,还有一个需要在进程下一次空闲时执行的代码队列。随着页面在其生命周期中的推移,代码会按照执行顺序添加入队列。例如,当某个按钮被按下时,它的事件处理程序代码就会被添加到队列中,并在下一个可能的时间里执行。当接收到某个 Ajax 响应时,回调函数的代码会被添加到队列。在 JavaScript 中没有任何代码是立刻执行的,但一旦进程空闲则尽快执行。

定时器对队列的工作方式是,当特定时间过去后将代码插入。注意,给队列添加代码并不意味着对它立刻执行,而只能表示它会尽快执行。设定一个 150ms 后执行的定时器不代表到了 150ms 代码就立刻执行,它表示代码会在 150ms 后被加入到队列中。如果在这个时间点上,队列中没有其他东西,那么这段代码就会被执行,表面上看上去好像代码就在精确指定的时间点上执行了。其他情况下,代码可能明显地等待更长时间才执行。

请看以下代码:

```
var btn = document.getElementById("my-btn");
btn.onclick = function(){
    setTimeout(function(){
        document.getElementById("message").style.visibility = "visible";
    }, 250);

    //其他代码
};
```

在这里给一个按钮设置了一个事件处理程序。事件处理程序设置了一个 250ms 后调用的定时器。点击该按钮后，首先将 onclick 事件处理程序加入队列。该程序执行后才设置定时器，再有 250ms 后，指定的代码才被添加到队列中等待执行。实际上，对 setTimeout() 的调用表示要晚点执行某些代码。

关于定时器要记住的最重要的事情是，指定的时间间隔表示何时将定时器的代码添加到队列，而不是何时实际执行代码。如果前面例子中的 onclick 事件处理程序执行了 300ms，那么定时器的代码至少要在定时器设置之后的 300ms 后才会被执行。队列中所有的代码都要等到 JavaScript 进程空闲之后才能执行，而不管它们是如何添加到队列中的。见图 22-2。

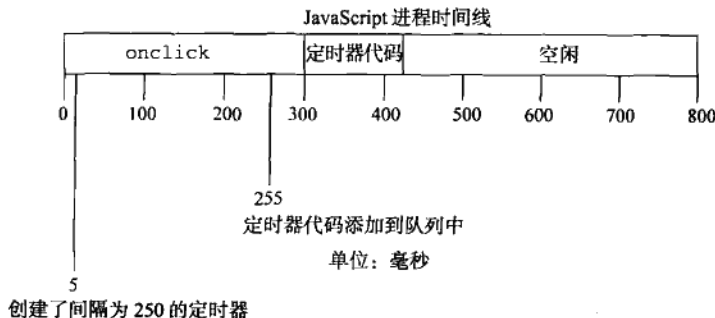


图 22-2

如图 22-2 所示，尽管在 255ms 处添加了定时器代码，但这时候还不能执行，因为 onclick 事件处理程序仍在运行。定时器代码最早能执行的时机是在 300ms 处，即 onclick 事件处理程序结束之后。

实际上 Firefox 中定时器的实现还能让你确定定时器过了多久才执行，这需传递一个实际执行的时间与指定的间隔的差值。如下面的例子所示。

```
//仅 Firefox 中
setTimeout(function(diff){
    if (diff > 0) {
        //晚调用
    } else if (diff < 0){
        //早调用
    } else {
        //调用及时
    }
}, 250);
```

执行完一套代码后，JavaScript 进程返回一段很短的时间，这样页面上的其他处理就可以进行了。由于 JavaScript 进程会阻塞其他页面处理，所以必须有这些小间隔来防止用户界面被锁定（代码长时间运行中还有可能出现）。这样设置一个定时器，可以确保在定时器代码执行前至少有一个进程间隔。

### 22.3.1 重复的定时器

使用 setInterval() 创建的定时器确保了定时器代码规则地插入队列中。这个方式的问题在于，定时器代码可能在代码再次被添加到队列之前还没有完成执行，结果导致定时器代码连续运行好几次，而之间没有任何停顿。幸好，JavaScript 引擎够聪明，能避免这个问题。当使用 setInterval() 时，仅当没有该定时器的任何其他代码实例时，才将定时器代码添加到队列中。这确保了定时器代码加入到队

列中的最小时间间隔为指定间隔。

这种重复定时器的规则有 2 点问题：(1) 某些间隔会被跳过；(2) 多个定时器的代码执行之间的间隔可能会比预期的小。假设，某个 onclick 事件处理程序使用 setInterval() 设置了一个 200ms 间隔的重复定时器。如果事件处理程序花了 300ms 多一点的时间完成，同时定时器代码也花了差不多的时间，就会跳过一个间隔同时运行着一个定时器代码。参见图 22-3。

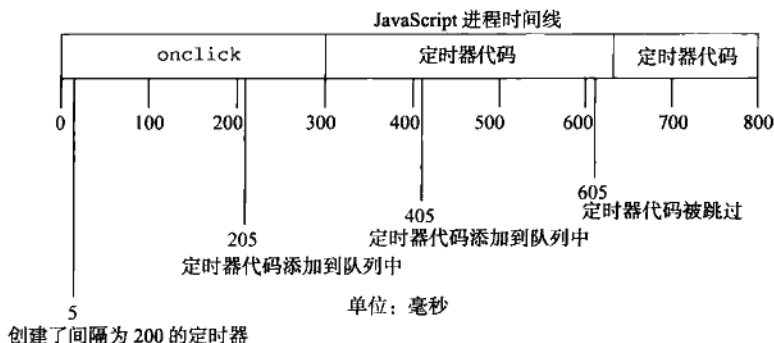


图 22-3

这个例子中的第 1 个定时器是在 205ms 处添加到队列中的，但是直到过了 300ms 处才能够执行。当执行这个定时器代码时，在 405ms 处又给队列添加了另外一个副本。在下一个间隔，即 605ms 处，第一个定时器代码仍在运行，同时在队列中已经有了一个定时器代码的实例。结果是，在这个时间点上的定时器代码不会被添加到队列中。结果在 5ms 处添加的定时器代码结束之后，405ms 处添加的定时器代码就立刻执行。

为了避免 setInterval() 的重复定时器的这 2 个缺点，你可以用如下模式使用链式 setTimeout() 调用。

```
setTimeout(function(){  
    //处理中  
    setTimeout(arguments.callee, interval);  
}, interval);
```

这个模式链式调用了 setTimeout()，每次函数执行的时候都会创建一个新的定时器。第二个 setTimeout() 调用使用了 arguments.callee 来获取对当前执行的函数的引用，并为其设置另外一个定时器。这样做的好处是，在前一个定时器代码执行完之前，不会向队列插入新的定时器代码，确保不会有任何缺失的间隔。而且，它可以保证在下一次定时器代码执行之前，至少要等待指定的间隔，避免了连续的运行。这个模式主要用于重复定时器，如下例所示。

```
setTimeout(function(){  
    var div = document.getElementById("myDiv");  
    left = parseInt(div.style.left) + 5;  
    div.style.left = left + "px";  
  
    if (left < 200){
```



```
        setTimeout(arguments.callee, 50);  
    }  
  
    }, 50);
```

这段定时器代码每次执行的时候将一个<div>元素向右移动，当左坐标在 200 像素的时候停止。JavaScript 动画中使用这个模式很常见。



每个浏览器窗口、标签页、或者框架都有其各自的代码执行队列。这意味着，进行跨框架或者跨窗口的定时调用，当代码同时执行的时候可能会导致竞争条件。无论何时需要使用这种通信类型，最好是在接收框架或者窗口中创建一个定时器来执行代码。

## 22.3.2 Yielding Processes

运行在浏览器中的 JavaScript 都被分配了一个确定数量的资源。不同于桌面应用往往能够随意控制他们要的内存大小和处理器时间，JavaScript 被严格限制了，以防止恶意的 Web 程序员把用户的计算机搞挂了。其中一个限制是长时间运行脚本的制约，如果代码运行超过特定的时间或者特定语句数量就不让它继续执行。如果代码达到了这个限制，会弹出一个浏览器错误的对话框，告诉用户某个脚本会用过长的时间执行，询问是允许其继续执行还是停止它。所有 JavaScript 开发人员的目标就是，确保用户永远不会在浏览器中看到这个令人费解的对话框。定时器是绕开此限制的方法之一。

脚本长时间运行的问题通常是由两个原因之一造成的：过长的、过深嵌套的函数调用或者是进行大量处理的循环。这两者中，后者是较为容易解决的问题。长时间运行的循环通常遵循以下模式：

```
for (var i=0, len=data.length; i < len; i++){  
    process(data[i]);  
}
```

这个模式的问题在于要处理的项目的数量在运行前是不可知的。如果完成 process() 要花 100ms，只有 2 个项目的数组可能不会造成影响，但是 10 个的数组可能会导致脚本要运行一秒钟才能完成。数组中的项目数量直接关系到执行完该循环的时间长度。同时由于 JavaScript 的执行是一个阻塞操作，脚本运行所花时间越久，用户无法与页面交互的时间也越久。

在展开该循环之前，你需要回答以下两个重要的问题。

- 该处理是否必须同步完成？如果这个数据的处理会造成其他运行的阻塞，那么最好不要改动它。不过，如果你对这个问题的回答确定为“否”，那么将某些处理推迟到以后是个不错的备选项。
- 数据是否必须按顺序完成？通常，数组只是对项目的组合和迭代的一种简便的方法而无所谓顺序。如果项目的顺序不是非常重要，那么可能可以将某些处理推迟到以后。

当你发现某个循环占用了大量时间，同时对于上述两个问题，你的回答都是“否”，那么你就可以使用定时器分割这个循环。这是一种叫做数组分块（array chunking）的技术，小块小块地处理数组，通常每次一小块。基本的思路是为要处理的项目创建一个队列，然后使用定时器取出下一个要处理的项目进行处理，接着再设置另一个定时器。基本的模式如下。

```


setTimeout(function(){

    //取出下一个条目并处理
    var item = array.shift();
    process(item);

    //若还有条目，再设置另一个定时器
    if(array.length > 0){
        setTimeout(arguments.callee, 100);
    }
}, 100);

```

在数组分块模式中，array 变量本质上就是一个“待办事宜”列表，它包含了要处理的项目。使用 shift() 方法可以获取队列中下一个要处理的项目，然后将其传递给某个函数。如果在队列中还有其他项目，则设置另一个定时器，并通过 arguments.callee 调用同一个匿名函数。要实现数组分块非常简单，可以使用以下函数。



```

function chunk(array, process, context){
    setTimeout(function(){
        var item = array.shift();
        process.call(context, item);


        if (array.length > 0){
            setTimeout(arguments.callee, 100);
        }
    }, 100);
}

```

---

*ArrayChunkingExample.htm*

chunk() 方法接受三个参数：要处理的项目的数组，用于处理项目的函数，以及可选的运行该函数的环境。函数内部用了之前描述过的基本模式，通过 call() 调用的 process() 函数，这样可以设置一个合适的执行环境（如果必须）。定时器的时间间隔设置为了 100ms，使得 JavaScript 进程有时间在处理项目的事件之间转入空闲。你可以根据你的需要更改这个间隔大小，不过 100ms 在大多数情况下效果不错。可以按如下所示使用该函数：



```

var data = [12,123,1234,453,436,23,23,5,4123,45,346,5634,2234,345,342];

function printValue(item){
    var div = document.getElementById("myDiv");
    div.innerHTML += item + "<br>";
}

chunk(data, printValue);

```

---

*ArrayChunkingExample.htm*

这个例子使用 printValue() 函数将 data 数组中的每个值输出到一个 <div> 元素。由于函数处在全局作用域内，因此无需给 chunk() 传递一个 context 对象。

必须当心的地方是，传递给 chunk() 的数组是用作一个队列的，因此当处理数据的同时，数组中的条目也在改变。如果你想保持原数组不变，则应该将该数组的克隆传递给 chunk()，如下例所示：

```

chunk(data.concat(), printValue);

```

当不传递任何参数调用某个数组的 `concat()` 方法时，将返回和原来数组中项目一样的数组。这样你就可以保证原数组不会被该函数更改。

数组分块的重要性在于它可以将多个项目的处理在执行队列上分开，在每个项目处理之后，给予其他的浏览器处理机会运行，这样就可能避免长时间运行脚本的错误。



一旦某个函数需要花 50ms 以上的时间完成，那么最好看看能否将任务分割为一系列可以使用定时器的小任务。

### 22.3.3 函数节流

浏览器中某些计算和处理要比其他的昂贵很多。例如，DOM 操作比起非 DOM 交互需要更多的内存和 CPU 时间。连续尝试进行过多的 DOM 相关操作可能会导致浏览器挂起，有时候甚至会崩溃。尤其在 IE 中使用 `onresize` 事件处理程序的时候容易发生，当调整浏览器大小的时候，该事件会连续触发。在 `onresize` 事件处理程序内部如果尝试进行 DOM 操作，其高频率的更改可能会让浏览器崩溃。为了绕开这个问题，你可以使用定时器对该函数进行节流。

函数节流背后的基本思想是指，某些代码不可以在没有间断的情况连续重复执行。第一次调用函数，创建一个定时器，在指定的时间间隔之后运行代码。当第二次调用该函数时，它会清除前一次的定时器并设置另一个。如果前一个定时器已经执行过了，这个操作就没有任何意义。然而，如果前一个定时器尚未执行，其实就是将其替换为一个新的定时器。目的是只有在执行函数的请求停止了一段时间之后才执行。以下是该模式的基本形式：

```
var processor = {
    timeoutId: null,

    //实际进行处理的方法
    performProcessing: function(){
        //实际执行的代码
    },

    //初始处理调用的方法
    process: function(){
        clearTimeout(this.timeoutId);


        var that = this;
        this.timeoutId = setTimeout(function(){
            that.performProcessing();
        }, 100);
    }
};

//尝试开始执行
processor.process();
```

在这段代码中，创建了一个叫做 `processor` 对象。这个对象还有 2 个方法：`process()` 和 `performProcessing()`。前者是初始化任何处理所必须调用的，后者则实际进行应完成的处理。当调用了 `process()`，第一步是清除存好的 `timeoutId`，来阻止之前的调用被执行。然后，创建一个新的定时器调用 `performProcessing()`。由于 `setTimeout()` 中用到的函数的环境总是 `window`，所以有必要保存 `this` 的引用以方便以后使用。

时间间隔设为了 100ms, 这表示最后一次调用 `process()` 之后至少 100ms 后才会调用 `performanceProcessing()`。所以如果 100ms 之内调用了 `process()` 共 20 次, `performanceProcessing()` 仍只会被调用一次。

这个模式可以使用 `throttle()` 函数来简化, 这个函数可以自动进行定时器的设置和清除, 如下例所示:



```
function throttle(method, context) {
    clearTimeout(method.tId);
    method.tId= setTimeout(function(){
        method.call(context);
    }, 100);
}
```


[ThrottlingExample.htm](#)

`throttle()` 函数接受两个参数: 要执行的函数以及在哪个作用域中执行。上面这个函数首先清除之前设置的任何定时器。定时器 ID 是存储在函数的 `tId` 属性中的, 第一次把方法传递给 `throttle()` 的时候, 这个属性可能并不存在。接下来, 创建一个新的定时器, 并将其 ID 储存在方法的 `tId` 属性中。如果这是第一次对这个方法调用 `throttle()` 的话, 那么这段代码会创建该属性。定时器代码使用 `call()` 来确保方法在适当的环境中执行。如果没有给出第二个参数, 那么就在全局作用域内执行该方法。

前面提到过, 节流在 `resize` 事件中是最常用的。如果你基于该事件来改变页面布局的话, 最好控制处理的频率, 以确保浏览器不会在极短的时间内进行过多的计算。例如, 假设有一个 `<div/>` 元素需要保持它的高度始终等同于宽度。那么实现这一功能的 JavaScript 可以如下编写:

```
window.onresize = function(){
    var div = document.getElementById("myDiv");
    div.style.height = div.offsetWidth + "px";
};
```

这段非常简单的例子有两个问题可能会造成浏览器运行缓慢。首先, 要计算 `offsetWidth` 属性, 如果该元素或者页面上其他元素有非常复杂的 CSS 样式, 那么这个过程将会很复杂。其次, 设置某个元素的高度需要对页面进行回流来令改动生效。如果页面有很多元素同时应用了相当数量的 CSS 的话, 这又需要很多计算。这就可以用到 `throttle()` 函数, 如下例所示:



```
function resizeDiv(){
    var div = document.getElementById("myDiv");
    div.style.height = div.offsetWidth + "px";
}

window.onresize = function(){
    throttle(resizeDiv);
};
```

[ThrottlingExample.htm](#)

这里, 调整大小的功能被放入了一个叫做 `resizeDiv()` 的单独函数中。然后 `onresize` 事件处理程序调用 `throttle()` 并传入 `resizeDiv` 函数, 而不是直接调用 `resizeDiv()`。多数情况下, 用户是感觉不到变化的, 虽然给浏览器节省的计算可能会非常大。

只要代码是周期性执行的，都应该使用节流，但是你不能控制请求执行的速率。这里展示的 `throttle()` 函数用了 100ms 作为间隔，你当然可以根据你的需要来修改它。

## 22.4 自定义事件

在本书前面，你已经学到事件是 JavaScript 与浏览器交互的主要途径。事件是一种叫做观察者的设计模式，这是一种创建松散耦合代码的技术。对象可以发布事件，用来表示在该对象生命周期中某个有趣的时刻到了。然后其他对象可以观察该对象，等待这些有趣的时刻到来并通过运行代码来响应。

观察者模式由两类对象组成：主体和观察者。主体负责发布事件，同时观察者通过订阅这些事件来观察该主体。该模式的一个关键概念是主体并不知道观察者的任何事情，也就是说它可以独自存在并正常运作即使观察者不存在。从另一方面来说，观察者知道主体并能注册事件的回调函数（事件处理程序）。涉及 DOM 上时，DOM 元素便是主体，你的事件处理代码便是观察者。

事件是与 DOM 交互的最常见的方式，但它们也可以用于非 DOM 代码中——通过实现自定义事件。自定义事件背后的概念是创建一个管理事件的对象，让其他对象监听那些事件。实现此功能的基本模式可以如下定义：

```
function EventTarget(){
    this.handlers = {};
}

EventTarget.prototype = {
    constructor: EventTarget,
    addHandler: function(type, handler){
        if (typeof this.handlers[type] == "undefined"){
            this.handlers[type] = [];
        }
        this.handlers[type].push(handler);
    },
    fire: function(event){
        if (!event.target){
            event.target = this;
        }
        if (this.handlers[event.type] instanceof Array){
            var handlers = this.handlers[event.type];
            for (var i=0, len=handlers.length; i < len; i++){
                handlers[i](event);
            }
        }
    },
    removeHandler: function(type, handler){
        if (this.handlers[type] instanceof Array){
            var handlers = this.handlers[type];
            for (var i=0, len=handlers.length; i < len; i++){
                if (handlers[i] == handler){
                    break;
                }
            }
            handlers.splice(i, 1);
        }
    }
}
```

```
}  
}  
};
```

## EventTarget.js

EventTarget 类型有一个单独的属性 handlers，用于储存事件处理程序。还有三个方法：addHandler()，用于注册给定类型事件的事件处理程序；fire()，用于触发一个事件；removeHandler()，用于注销某个事件类型的事件处理程序。

addHandler() 方法接受两个参数：事件类型和用于处理该事件的函数。当调用该方法时，会进行一次检查，看看 handlers 属性中是否已经存在一个针对该事件类型的数组；如果没有，则创建一个新的。然后使用 push() 将该处理程序添加到数组的末尾。

如果要触发一个事件，要调用 fire() 函数。该方法接受一个单独的参数，是一个至少包含 type 属性的对象。fire() 方法先给 event 对象设置一个 target 属性，如果它尚未被指定的话。然后它就查找对应该事件类型的一组处理程序，调用各个函数，并给出 event 对象。因为这些都是自定义事件，所以 event 对象上还需要的额外信息由你自己决定。

removeHandler() 方法是 addHandler() 的辅助，它们接受的参数一样：事件的类型和事件处理程序。这个方法搜索事件处理程序的数组找到要删除的处理程序的位置。如果找到了，则使用 break 操作符退出 for 循环。然后使用 splice() 方法将该项目从数组中删除。

然后，使用 EventTarget 类型的自定义事件可以如下使用：



```
function handleMessage(event){  
    alert("Message received: " + event.message);  
}  
  
//创建一个新对象  
var target = new EventTarget();  
  
//添加一个事件处理程序  
target.addHandler("message", handleMessage);  
  
//触发事件  
target.fire({ type: "message", message: "Hello world!"});  
  
//删除事件处理程序  
target.removeHandler("message", handleMessage);  
  
//再次，应没有处理程序  
target.fire({ type: "message", message: "Hello world!"});
```

## EventTargetExample01.htm

在这段代码中，定义了 handleMessage() 函数用于处理 message 事件。它接受 event 对象并输出 message 属性。调用 target 对象的 addHandler() 方法并传给 "message" 以及 handleMessage() 函数。在接下来的一行上，调用了 fire() 函数，并传递了包含 2 个属性，即 type 和 message 的对象直接量。它会调用 message 事件的事件处理程序，这样就会显示一个警告框（来自 handleMessage()）。然后删除了事件处理程序，这样即使事件再次触发，也不会显示任何警告框。

因为这种功能是封装在一种自定义类型中的，其他对象可以继承 EventTarget 并获得这个行为，

如下例所示：

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

inheritPrototype(Person, EventTarget);

Person.prototype.say = function(message){
    this.fire({type: "message", message: message});
};
```

---

*EventTargetExample02.htm*

Person 类型使用了寄生组合继承（参见第 6 章）方法来继承 EventTarget。一旦调用了 say() 方法，便触发了事件，它包含了消息的细节。在某种类型的另外的方法中调用 fire() 方法是很常见的，同时它通常不是公开调用的。这段代码可以照如下方式使用：

```
function handleMessage(event){
    alert(event.target.name + " says: " + event.message);
}

//创建新 person
var person = new Person("Nicholas", 29);

//添加一个事件处理程序
person.addHandler("message", handleMessage);

//在该对象上调用 1 个方法，它触发消息事件
person.say("Hi there.");
```

---

*EventTargetExample02.htm*

这个例子中的 handleMessage() 函数显示了某人名字（通过 event.target.name 获得）的一个警告框和消息正文。当调用 say() 方法并传递一个消息时，就会触发 message 事件。接下来，它又会调用 handleMessage() 函数并显示警告框。

当代码中存在多个部分在特定时刻相互交互的情况下，自定义事件就非常有用。这时，如果每个对象都有对其他所有对象的引用，那么整个代码就会紧密耦合，同时维护也变得很困难，因为对某个对象的修改也会影响到其他对象。使用自定义事件有助于解耦相关对象，保持功能的隔绝。在很多情况中，触发事件的代码和监听事件的代码是完全分离的。

## 22.5 拖放

拖放是一种非常流行的用户界面模式。它的概念很简单：点击某个对象，并按住鼠标按钮不放，将鼠标移动到另一个区域，然后释放鼠标按钮将对象“放”在这里。拖放功能也流行到了 Web 上，成为了一些更传统的配置界面的一种候选方案。

拖放的基本概念很简单：创建一个绝对定位的元素，使其可以用鼠标移动。这个技术源自一种叫做

“鼠标拖尾”的经典网页技巧。鼠标拖尾是一个或者多个图片在页面上跟着鼠标指针移动。单元素鼠标拖尾的基本代码需要为文档设置一个 `onmousemove` 事件处理程序，它总是将指定元素移动到鼠标指针的位置，如下面的例子所示。



```
EventUtil.addHandler(document, "mousemove", function(event){
    var myDiv = document.getElementById("myDiv");
    myDiv.style.left = event.clientX + "px";
    myDiv.style.top = event.clientY + "px";
});
```

[DragAndDropExample01.htm](#)

在这个例子中，元素的 `left` 和 `top` 坐标设置为了 `event` 对象的 `clientX` 和 `clientY` 属性，这就将元素放到了视口中指针的位置上。它的效果是一个元素始终跟随指针在页面上的移动。只要正确的时刻（当鼠标按钮按下的时候）实现该功能，并在之后删除它（当释放鼠标按钮时），就可以实现拖放了。最简单的拖放界面可用以下代码实现：

```
var DragDrop = function(){

    var dragging = null;

    function handleEvent(event){

        //获取事件和目标
        event = EventUtil.getEvent(event);
        var target = EventUtil.getTarget(event);

        //确定事件类型
        switch(event.type){
            case "mousedown":
                if (target.className.indexOf("draggable") > -1){
                    dragging = target;
                }
                break;

            case "mousemove":
                if (dragging !== null){

                    //assign location
                    dragging.style.left = event.clientX + "px";
                    dragging.style.top = event.clientY + "px";
                }
                break;

            case "mouseup":
                dragging = null;
                break;
        }
    }

    //公共接口
    return {
        enable: function(){
            EventUtil.addHandler(document, "mousedown", handleEvent);
            EventUtil.addHandler(document, "mousemove", handleEvent);
            EventUtil.addHandler(document, "mouseup", handleEvent);
        },
    },
};
```



```

disable: function(){
    EventUtil.removeHandler(document, "mousedown", handleEvent);
    EventUtil.removeHandler(document, "mousemove", handleEvent);
    EventUtil.removeHandler(document, "mouseup", handleEvent);
}
}
})();

```

## DragAndDropExample02.htm

DragDrop 对象封装了拖放的所有基本功能。这是一个单例对象，并使用了模块模式来隐藏某些实现细节。dragging 变量起初是 null，将会存放被拖动的元素，所以当该变量不为 null 时，就知道正在拖动某个东西。handleEvent() 函数处理拖放功能中的所有的三个鼠标事件。它首先获取 event 对象和事件目标的引用。之后，用一个 switch 语句确定要触发哪个事件样式。当 mousedown 事件发生时，会检查 target 的 class 是否包含 "draggable" 类，如果是，那么将 target 存放到 dragging 中。这个技巧可以很方便地通过标记语言而非 JavaScript 脚本来确定可拖动的元素。

handleEvent() 的 mousemove 情况和前面的代码一样，不过要检查 dragging 是否为 null。当它不是 null，就知道 dragging 就是要拖动的元素，这样就会把它放到恰当的位置上。mouseup 情况就仅仅是将 dragging 重置为 null，让 mousemove 事件中的判断失效。

DragDrop 还有两个公共方法：enable() 和 disable()，它们只是相应添加和删除所有的事件处理程序。这两个函数提供了额外的对拖放功能的控制手段。

要使用 DragDrop 对象，只要在页面上包含这些代码并调用 enable()。拖放会自动针对所有包含 "draggable" 类的元素启用，如下例所示：

```
<div class="draggable" style="position:absolute; background:red"> </div>
```

注意为了元素能被拖放，它必须是绝对定位的。

## 22.5.1 修缮拖动功能

当你试了上面的例子之后，你会发现元素的左上角总是和指针在一起。这个结果对用户来说有一点不爽，因为当鼠标开始移动的时候，元素好像是突然跳了一下。理想情况是，这个动作应该看上去好像这个元素是被指针“拾起”的，也就是说当在拖动元素的时候，用户点击的那一点就是指针应该保持的位置（见图 22-4）。



用户首先点击的是这里



被拖动后，指针就跑到了这里了

图 22-4

要达到需要的效果，必须做一些额外的计算。你需要计算元素左上角和指针位置之间的差值。这个差值应该在 mousedown 事件发生的时候确定，并且一直保持，直到 mouseup 事件发生。通过将 event 的 clientX 和 clientY 属性与该元素的 offsetLeft 和 offsetTop 属性进行比较，就可以算出水平

方向和垂直方向上需要多少空间, 见图 22-5。

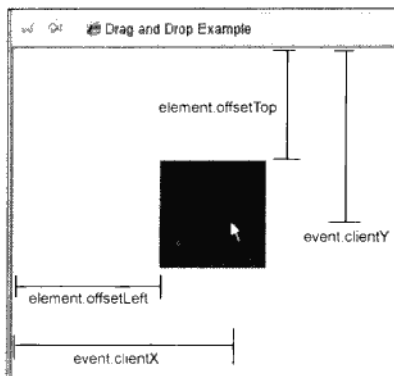


图 22-5

为了保存 x 和 y 坐标上的差值, 还需要几个变量。diffX 和 diffY 这些变量需要在 onmousemove 事件处理程序中用到, 来对元素进行适当的定位, 如下面的例子所示。

```
var DragDrop = function(){  
    var dragging = null;  
    diffX = 0;  
    diffY = 0;  
  
    function handleEvent(event){  
        // 获取事件和目标  
        event = EventUtil.getEvent(event);  
        var target = EventUtil.getTarget(event);  
  
        // 确定事件类型  
        switch(event.type){  
            case "mousedown":  
                if (target.className.indexOf("draggable") > -1){  
                    dragging = target;  
                    diffX = event.clientX - target.offsetLeft;  
                    diffY = event.clientY - target.offsetTop;  
                }  
                break;  
  
            case "mousemove":  
                if (dragging !== null){  
                    // 指定位置  
                    dragging.style.left = (event.clientX - diffX) + "px";  
                    dragging.style.top = (event.clientY - diffY) + "px";  
                }  
                break;  
  
            case "mouseup":
```

```

        dragging = null;
        break;
    }
};

//公共接口
return {
    enable: function(){
        EventUtil.addHandler(document, "mousedown", handleEvent);
        EventUtil.addHandler(document, "mousemove", handleEvent);
        EventUtil.addHandler(document, "mouseup", handleEvent);
    },
    disable: function(){
        EventUtil.removeHandler(document, "mousedown", handleEvent);
        EventUtil.removeHandler(document, "mousemove", handleEvent);
        EventUtil.removeHandler(document, "mouseup", handleEvent);
    }
}
})();

```

---

*DragAndDropExample03.htm*

diffX 和 diffY 变量是私有的，因为只有 handleEvent() 函数需要用到它们。当 mousedown 事件发生时，通过 clientX 减去目标的 offsetLeft，clientY 减去目标的 offsetTop，可以计算到这两个变量的值。当触发了 mousemove 事件后，就可以使用这些变量从指针坐标中减去，得到最终的坐标。最后得到一个更加平滑的拖动体验，更加符合用户所期望的方式。

## 22.5.2 添加自定义事件

拖放功能还不能真正应用起来，除非能知道什么时候拖动开始了。从这点上看，前面的代码没有提供任何方法表示拖动开始、正在拖动或者已经结束。这时，可以使用自定义事件来指示这几个事件的发生，让应用的其他部分与拖动功能进行交互。

由于 DragDrop 对象是一个使用了模块模式的单例，所以需要进行一些更改来使用 EventTarget 类型。首先，创建一个新的 EventTarget 对象，然后添加 enable() 和 disable() 方法，最后返回这个对象。看以下内容。

```

var DragDrop = function(){
    var dragdrop = new EventTarget(),
        dragging = null,
        diffX = 0,
        diffY = 0;

    function handleEvent(event){
        //获取事件和对象
        event = EventUtil.getEvent(event);
        var target = EventUtil.getTarget(event);

        //确定事件类型

```

```

switch(event.type){
    case "mousedown":
        if (target.className.indexOf("draggable") > -1){
            dragging = target;
            diffX = event.clientX - target.offsetLeft;
            diffY = event.clientY - target.offsetTop;
            dragdrop.fire({type:"dragstart", target: dragging,
                           x: event.clientX, y: event.clientY});
        }
        break;

    case "mousemove":
        if (dragging !== null){
            //指定位置
            dragging.style.left = (event.clientX - diffX) + "px";
            dragging.style.top = (event.clientY - diffY) + "px";

            //触发自定义事件
            dragdrop.fire({type:"drag", target: dragging,
                           x: event.clientX, y: event.clientY});
        }
        break;

    case "mouseup":
        dragdrop.fire({type:"dragend", target: dragging,
                       x: event.clientX, y: event.clientY});
        dragging = null;
        break;
}
};

//公共接口
dragdrop.enable = function(){
    EventUtil.addHandler(document, "mousedown", handleEvent);
    EventUtil.addHandler(document, "mousemove", handleEvent);
    EventUtil.addHandler(document, "mouseup", handleEvent);
};

dragdrop.disable = function(){
    EventUtil.removeHandler(document, "mousedown", handleEvent);
    EventUtil.removeHandler(document, "mousemove", handleEvent);
    EventUtil.removeHandler(document, "mouseup", handleEvent);
};

return dragdrop;
}();

```

*[DragAndDropExample04.htm](#)*

这段代码定义了三个事件: dragstart、drag 和 dragend。它们都将被拖动的元素设置为了 target, 并给出了 x 和 y 属性来表示当前的位置。它们触发于 dragdrop 对象上, 之后在返回对象前给对象增加 enable() 和 disable() 方法。这些模块模式中的细小更改令 DragDrop 对象支持了事件, 如下:



```

DragDrop.addHandler("dragstart", function(event){
    var status = document.getElementById("status");
    status.innerHTML = "Started dragging " + event.target.id;
});

DragDrop.addHandler("drag", function(event){
    var status = document.getElementById("status");
    status.innerHTML += "<br/> Dragged " + event.target.id + " to (" + event.x +
        "," + event.y + ")";
});

DragDrop.addHandler("dragend", function(event){
    var status = document.getElementById("status");
    status.innerHTML += "<br/> Dropped " + event.target.id + " at (" + event.x +
        "," + event.y + ")";
});

```

*DragAndDropExample04.htm*

这里，为 DragDrop 对象的每个事件添加了事件处理程序。还使用了一个元素来实现被拖动的元素当前的状态和位置。一旦元素被放下了，就可以看到从它一开始被拖动之后经过的所有的中间步骤。

为 DragDrop 添加自定义事件可以使这个对象更健壮，它将可以在网络应用中处理复杂的拖放功能。

## 22.6 小结

JavaScript 中的函数非常强大，因为它们是第一类对象。使用闭包和函数环境切换，还可以有很多使用函数的强大方法。可以创建作用域安全的构造函数，确保在缺少 new 操作符时调用构造函数不会改变错误的环境对象。

- 可以使用惰性载入函数，将任何代码分支推迟到第一次调用函数的时候。
  - 函数绑定可以让你创建始终在指定环境中运行的函数，同时函数柯里化可以让你创建已经填了某些参数的函数。
  - 将绑定和柯里化组合起来，就能够给你一种在任意环境中以任意参数执行任意函数的方法。
- ECMAScript 5 允许通过以下几种方式来创建防篡改对象。
- 不可扩展的对象，不允许给对象添加新的属性或方法。
  - 密封的对象，也是不可扩展的对象，不允许删除已有的属性和方法。
  - 冻结的对象，也是密封的对象，不允许重写对象的成员。

JavaScript 中可以使用 setTimeout() 和 setInterval() 如下创建定时器。

- 定时器代码是放在一个等待区域，直到时间间隔到了之后，此时将代码添加到 JavaScript 的处理队列中，等待下一次 JavaScript 进程空闲时被执行。
- 每次一段代码执行结束之后，都会有一小段空闲时间进行其他浏览器处理。
- 这种行为意味着，可以使用定时器将长时间运行的脚本切分为一小块一小块可以在以后运行的代码段。这种做法有助于 Web 应用对用户交互有更积极的响应。

JavaScript 中经常以事件的形式应用观察者模式。虽然事件常常和 DOM 一起使用，但是你也可以通

过实现自定义事件在自己的代码中应用。使用自定义事件有助于将不同部分的代码相互之间解耦，让维护更加容易，并减少引入错误的机会。

拖放对于桌面和 Web 应用都是一个非常流行的用户界面范例，它能够让用户非常方便地以一种直观的方式重新排列或者配置东西。在 JavaScript 中可以使用鼠标事件和一些简单的计算来实现这种功能类型。将拖放行为和自定义事件结合起来可以创建一个可重复使用的框架，它能应用于各种不同的情况下。