**22高级技巧**

**高级函数**

1. 安全的类型检测

typeof 操作符，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。

instanceof 操作符在存在多个全局作用域（像一个页面包含多个 frame）的情况下，以下代码要返回 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。）

var isArray = value instanceof Array;

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。

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。

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

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]";

1. 作用域安全的构造函数

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，。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。[Demo1](demo1.html)

由于 window 的 name 属性是用于识别链接目标和 frame 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。

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);**

**}**

}

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。[Demo2](demo2.html)

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。[Demo2](demo2.html)

1. 懒性载入函数

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。[Demo3](demo3.html)

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。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（重新定义函数）。但第一次调用函数时会损失性能[Demo3](file:///E:\Mark\learn-mark\js高设\第二十二章高级技巧\demo3.html)

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。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。（第一次运行之后return function）[Demo3](file:///E:\Mark\learn-mark\js高设\第二十二章高级技巧\demo3.html)

1. 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。

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。this 对象最后是指向了 DOM 按钮而非 handler（在 IE8 中，this 指向 window。）

var handler = {

message: "Event handled",

handleClick: function(event){

alert(this.message);

}

};

var btn = document.getElementById("my-btn");

EventUtil.addHandler(btn, "click", handler.handleClick);

使用一个闭包来修正这个问题。

EventUtil.addHandler(btn, "click", function(event){

handler.handleClick(event);

});

创建多个闭包可能会令代码变得难于理解和调试。因此，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);

};

}

EventUtil.addHandler(btn, "click", bind(handler.handleClick, handler));

1. 函数柯里化

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);

};

}

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);

};

}

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"));

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"));

**防篡改对象**

为任何对象都可以被在同一环境中运行的代码修改。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）。

1. 不可扩展对象

默认情况下，所有对象都是可以扩展的。

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。[Demo4](demo4.html)

Object.preventExtensions(person);

1. 密封对象

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。**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" };

alert(Object.isExtensible(person)); //true

alert(Object.isSealed(person)); //false

Object.seal(person);

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

alert(Object.isSealed(person)); //true

delete person.name;//严格模式下尝试添加或删除对象成员都会导致抛出错误。

1. 冻结对象

最严格的防篡改级别是冻结对象（frozen object）。

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

person.age = 29;

delete person.name;

person.name = "Greg";//严格模式下修改都报错

**高级定时器**

var btn = document.getElementById("my-btn");

btn.onclick = function(){

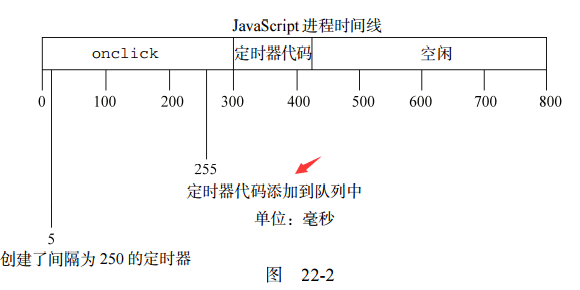
setTimeout(function(){

document.getElementById("message").style.visibility = "visible";

}, 250);

//其他代码

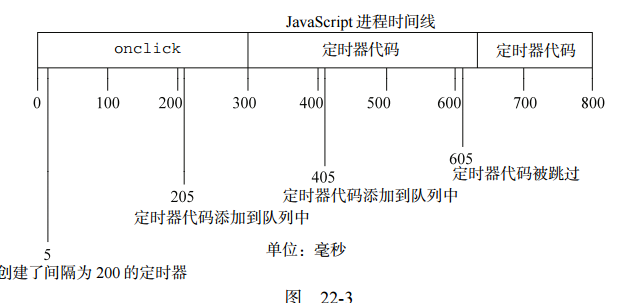
};



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。

1. 重复定时器

重复定时器的规则有两个问题：(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。



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。

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。

1. Yielding（让步；可伸缩性） Processes

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。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]);

}

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。

q 该处理是否必须同步完成？

q 数据是否必须按顺序完成？

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。**基本的模式如下。

function chunk(array, process, context){

setTimeout(function(){

var item = array.shift();

process.call(context, item);

if (array.length > 0){

setTimeout(arguments.callee, 100);

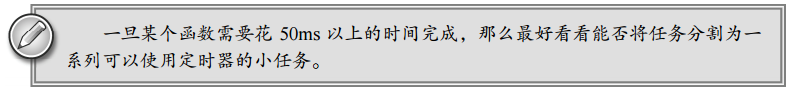
}

}, 100);

}

必须当心的地方是，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。



1. 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。尤其在 IE 中调整浏览器大小的时候，onresize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**。

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();

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 performProcessing()。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);

}

**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。

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);

}

}

};

**拖放**

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。 [Demo5](demo4.html)