### 17.9.11 使用 this

this 用法比较灵活，它可以存在于任何位置，它并不仅仅局限于对象的方法内，还可以被应用在全局域内、函数内，以及其他特殊上下文环境中。

【示例 1】函数的引用和调用。

函数的引用和调用分别表示不同的概念。虽然它们都无法改变函数的定义作用域。但是引用函数，却能够改变函数的执行作用域，而调用函数是不会改变函数的执行作用域的。

```
var o = {
    name : "对象 o",
    f : function(){
        return this;
    }
}
o.o1 = {
    name : "对象 o1",
    me : o.f                    // 引用对象 o 的方法 f
}
```

在上面示例中，函数中的 this 所代表的是当前执行域对象 o1：

```
var who = o.o1.me();
alert(who.name);                // 返回字符串"对象 o1"，说明当前 this 代表对象 o1
```

如果把对象 o1 的 me 属性值改为函数调用：

```
o.o1 = {
    name : "对象 o1",
    me : o.f()                  // 调用对象 o 的方法 f
}
```

则函数中的 this 所代表的是定义函数时所在的作用域对象 o：

```
var who = o.o1.me;
alert(who.name);                // 返回字符串"对象 o"，说明当前 this 代表对象 o
```

【示例 2】使用 call()和 apply() 。

call()和 apply()方法可以直接改变被执行函数的作用域，使其作用域指向所传递的参数对象。因此，函数中包含的 this 关键字也指向参数对象。

```
function f(){
    // 如果当前执行域对象的构造函数等于当前函数，则表示 this 为实例对象
    if(this.constructor == arguments.callee) alert("this = 实例对象");
    // 如果当前执行域对象等于 window，则表示 this 为 Window 对象
    else if (this == window) alert("this = window 对象");
    // 如果当前执行域对象为其他对象，则表示 this 为其他对象
    else alert("this == 其他对象 \nthis.constructor = " +
          this.constructor );
}
f();                            // this 指向 Window 对象
new f();                        // this 指向实例对象
f.call(1);                      // this 指向数值实例对象
```

在上面示例中，直接调用函数 f()时，函数的执行作用域为全局域，所以 this 代表 window。当使用 new 运算符调用函数时，将创建一个新的实例对象，函数的执行作用域为实例对象所在的上下文，所以 this 就指向这个新创建的实例对象。

而使用 call()方法执行函数 f()时，call 会把函数 f()的作用域强制修改为参数对象所在的上下文。由于 call()方法的参数值为数字 1，则 JavaScript 解释器会把数字 1 强制封装为数值对象，此时 this 就会指向这个数值对象。

在下面示例中，call()方法把函数 f()强制转换为对象 o 的一个方法并执行，这样函数 f()中的 this 就指代对象 o，所以 this.x 的值就等于 1，而 this.y 的值就等于 2，结果就返回 3：

```
function f(){                    // 函数 f()
    alert(this.x + this.y);
}
var o = {                        // 对象直接量
    x : 1,
    y : 2
}
f.call(o);                       // 执行函数 f()，返回值为 3
```

**【示例 3】**原型继承。

JavaScript 通过原型模式实现类的延续和继承，那么在父类的成员中包含了 this 关键字时，当子类继承了父类的这些成员时，this 的指向就变得很迷惑人。

在一般情况下，子类继承父类的方法后，this 会指向子类的实例对象，但是也可能指向子类的原型对象，而不是子类的实例对象。

```
function Base(){                 // 基类
    this.m = function(){         // 基类的方法 m()
        return "Base";
    };
    this.a = this.m();           // 基类的属性 a，调用当前作用域中 m()方法
    this.b = this.m;             // 基类的方法 b()，引用当前作用域中 m()方法
    this.c = function(){
    // 基类的方法 c()，以闭包结构用当前作用域中 m()方法
        return this.m();
    }
}
function F(){                    // 子类
    this.m = function(){         // 子类的方法 m()
        return "F"
    }
}
F.prototype = new Base();        // 继承基类
var f = new F();                 // 实例化子类
alert(f.a);                      // 返回字符串"Base"，说明 this.m()中 this 指向 F 的原型对象
alert(f.b());                    // 返回字符串"Base"，说明 this.m()中 this 指向 F 的原型对象
alert(f.c());                    // 返回字符串"F"，说明 this.m()中 this 指向 F 的实例对象
```

在上面示例中，基类 Base 包含 4 个成员，其中成员 b 和 c 以不同方式引用当前作用域内方法 m()，而成员 a 存储着当前作用域内方法 m()的调用值。当这些成员继承给子类 F 后，其中 m、b 和 c 成为原型对象的方法，而 a 成为原型对象的属性。但是，c 的值为一个闭包体，当在子类的实例中调用时，实际上它的返回值已经成为实例对象的成员，也就是说，闭包体在哪儿被调用，则其中包含的 this 就会指向哪儿。所以，你会看到 f.c()中的 this 指向实例对象，而不是 F 类的原型对象。

为了避免因继承关系而影响父类中 this 所代表的对象，除了通过上面介绍的方法，把方法的引用传递给父类的成员外，我们还可以为父类定义私有函数，然后再把它的引用传递给其他父类成员，这样就避免了因为函数闭包的原因，而改变 this 的值。

```
function Base(){
    var _m = function(){                // 定义基类的私有函数_m()
        return "Base";
    };
    this.a = _m;
    this.b = _m();
}
```

这样基类的私有函数_m()就具有完全隐私性，外界其他任何对象都无法直接访问基类的私有函数_m()。所以，在一般情况下，定义方法的时候，对于相互依赖的方法，可以把它定义私有函数，并以引用方法的方式对外公开，这样就避免了外界对于依赖方法的影响。

【示例 4】异步调用之事件处理函数。

异步调用就是通过事件机制或者计时器来延迟函数的调用时间和时机。通过调用函数的执行作用域不再是原来的定义作用域，所以函数中的 this 总是指向引发该事件的对象。

```
<input type="button" value="Button" />
<script language="javascript" type="text/javascript">
var button = document.getElementsByTagName("input")[0];
var o ={};
o.f = function(){
    if(this == o) alert("this = o");
    if(this == window) alert("this = window");
    if(this == button) alert("this = button");
}
button.onclick = o.f;
</script>
```

这里的方法 f()所包含的 this 不再指向对象 o，而是指向按钮 button，因为它是被传递给按钮的事件处理函数之后，再被调用执行的。函数的执行作用域发生了变化，所以不再指向定义方法时所指定的对象。

如果使用 DOM 2 级标准为按钮注册事件处理函数：

```
if(window.attachEvent){            // 兼容 IE
    button.attachEvent("onclick", o.f);
}
else{                              // 兼容符合 DOM 标准的浏览器
    button.addEventListener("click", o.f, true);
}
```

则在 IE 浏览器中，this 指向 Window 和 button，而在符合 DOM 标准的浏览器中仅指向 button。因为，在 IE 浏览器中，attachEvent()是 Window 对象的方法，调用该方法时，执行作用域为全局作用域，所以 this 会指向 window。同时由于该方法被注册到按钮对象上，所以它的真正执行作用域应该为 button 对象所在的上下文。这一点可以通过在符合 DOM 标准的浏览器中看到。这种解释可能很勉强，但是在 IE 中 this 同时指向 Window 和 button 对象本身就让人迷惑不解。

为了解决这个问题，可以借助 call()或 apply()方法强制在对象 o 身上执行方法 f()，也就是说，强制改变 f()方法的执行作用域，避免因为环境的不同而影响函数作用域的变化。

```
if(window.attachEvent){
    button.attachEvent("onclick", function(){
   // 以闭包的形式封装 call()方法强制执行 f()
        o.f.call(o);
    });
}
else{
    button.addEventListener("click", function(){
        o.f.call(o);
    }, true);
}
```

这样当再次执行时，方法 f()中包含的 this 关键字始终指向对象 o，也就是说，它的执行作用域始终与它的定义作用域保持一致。

【示例 5】异步调用之定时器。

异步调用的另一种形式，就是使用定时器来调用函数，定时器就是指调用 window 对象的 setTimeout()或 setInterval()方法来延期调用函数。

例如，下面示例设计延期调用方法 o.f()。

```
var o ={};
o.f = function(){
    if(this == o) alert("this = o");
    if(this == window) alert("this = window");
```

```
        if(this == button) alert("this = button");
    }
    setTimeout(o.f, 100);
```

此时，经测试程序，会发现在 IE 中 this 指向 Window 和 Button 对象，具体原因与上面讲解的 attachEvent()方法相同。但是，在符合 DOM 标准的浏览器中，this 指向 Window 对象，而不是 Button 对象，因为方法 setTimeout()是在全局作用域中被执行的，所以 this 自然指向 Window 对象。要解决这个问题，仍然可以使用 call()或 apply()方法来实现：

```
    setTimeout(function(){
        o.f.call(o);
    }, 100);
```