# 第 6 章　代理模式

代理模式是为一个对象提供一个代用品或占位符，以便控制对它的访问。

代理模式是一种非常有意义的模式，在生活中可以找到很多代理模式的场景。比如，明星都有经纪人作为代理。如果想请明星来办一场商业演出，只能联系他的经纪人。经纪人会把商业演出的细节和报酬都谈好之后，再把合同交给明星签。

代理模式的关键是，当客户不方便直接访问一个对象或者不满足需要的时候，提供一个替身对象来控制对这个对象的访问，客户实际上访问的是替身对象。替身对象对请求做出一些处理之后，再把请求转交给本体对象。如图6-1和图6-2所示。

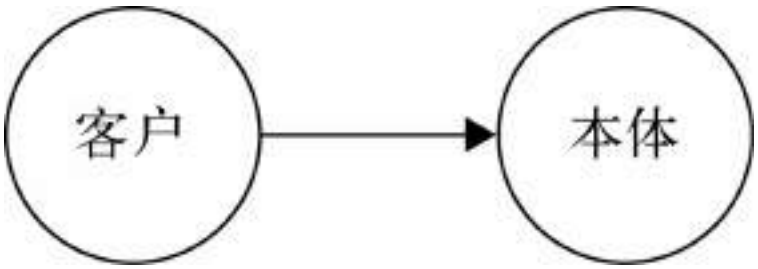



图 6-1　不用代理模式

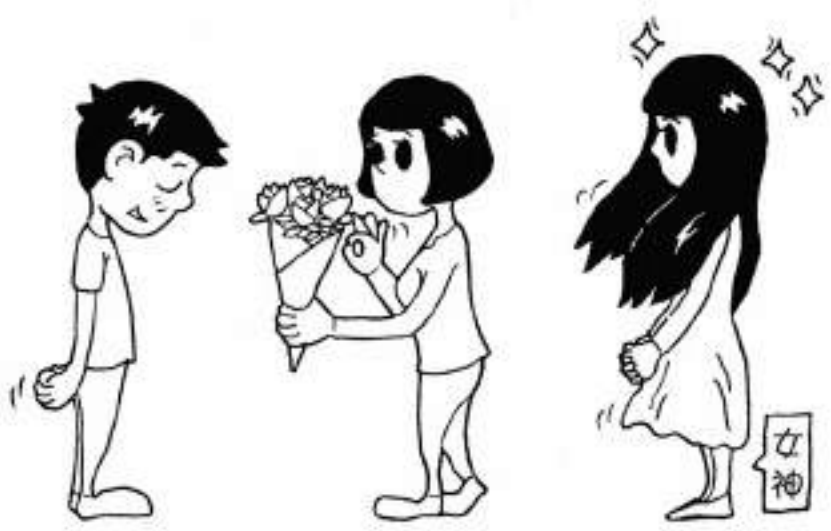

图 6-2　使用代理模式

下面我们通过几个例子来详细说明。

## 6.1　第一个例子——小明追MM的故事

下面我们从一个小例子开始熟悉代理模式的结构。

在四月一个晴朗的早晨，小明遇见了他的百分百女孩，我们暂且称呼小明的女神为A。两天之后，小明决定给A送一束花来表白。刚好小明打听到A和他有一个共同的朋友B，于是内向的小明决定让B来代替自己完成送花这件事情。




虽然小明的故事必然以悲剧收场，因为追MM更好的方式是送一辆宝马。不管怎样，我们还是先用代码来描述一下小明追女神的过程，先看看不用代理模式的情况：

```
var Flower = function(){};

var xiaoming = {
    sendFlower: function( target ){
        var flower = new Flower();
        target.receiveFlower( flower );
    }
};

var A = {
    receiveFlower: function( flower ){
        console.log( '收到花 ' + flower );
```

```
    }
};

xiaoming.sendFlower( A );
```

接下来，我们引入代理B，即小明通过B来给A送花：

```
var Flower = function(){};

var xiaoming = {
    sendFlower: function( target){
    var flower = new Flower();
        target.receiveFlower( flower );
    }
};

var B = {
    receiveFlower: function( flower ){
        A.receiveFlower( flower );
    }
};

var A = {
    receiveFlower: function( flower ){
        console.log( '收到花 ' + flower );
    }
};

xiaoming.sendFlower( B );
```

很显然，执行结果跟第一段代码一致，至此我们就完成了一个最简单的代理模式的编写。

也许读者会疑惑，小明自己去送花和代理B帮小明送花，二者看起来并没有本质的区别，引入一个代理对象看起来只是把事情搞复杂了而已。

的确，此处的代理模式毫无用处，它所做的只是把请求简单地转交给本体。但不管怎样，我们开始引入了代理，这是一个不错的起点。

现在我们改变故事的背景设定，假设当A在心情好的时候收到花，小明表白成功的几率有60%，而当A在心情差的时候收到花，小明表白的成功率无限趋近于0。

小明跟A刚刚认识两天，还无法辨别A什么时候心情好。如果不合时宜地把花送给A，花被直接扔掉的可能性很大，这束花可是小明吃了7天泡面换来的。

但是A的朋友B却很了解A，所以小明只管把花交给B，B会监听A的心情变化，然后选择A心情好的时候把花转交给A，代码如下：

```
var Flower = function(){};

var xiaoming = {
    sendFlower: function( target){
        var flower = new Flower();
        target.receiveFlower( flower );
    }
};

var B = {
    receiveFlower: function( flower ){
        A.listenGoodMood(function(){    // 监听A的好心情
            A.receiveFlower( flower );
        });
    }
};

var A = {
    receiveFlower: function( flower ){
        console.log( '收到花 ' + flower );
    },
    listenGoodMood: function( fn ){
        setTimeout(function(){    // 假设10秒之后A的心情变好
            fn();
        }, 10000 );
    }
};

xiaoming.sendFlower( B );
```

## 6.2 保护代理和虚拟代理

虽然这只是个虚拟的例子，但我们可以从中找到两种代理模式的身影。代理B可以帮助A过滤掉一些请求，比如送花的人中年龄太大的或者没有宝马的，这种请求就可以直接在代理B处被拒绝掉。这种代理叫作保护代理 。A和B一个充当白脸，一个充当黑脸。白脸A继续保持良好的女神形象，不希望直接拒绝任何人，于是找了黑脸B来控制对A的访问。

另外，假设现实中的花价格不菲，导致在程序世界里，new Flower 也是一个代价昂贵的操作，那么我们可以把new Flower 的操作交给代理B去执行，代理B会选择在A心情好时再执行new Flower ，这是代理模式的另一种形式，叫作虚拟代理 。虚拟代理把一些开销很大的对象，延迟到真正需要它的时候才去创建。代码如下：

```
var B = {
    receiveFlower: function( flower ){
        A.listenGoodMood(function(){     // 监听A的好心情
             var flower = new Flower();     // 延迟创建flower 对象
             A.receiveFlower( flower );
        });
    }
};
```

保护代理用于控制不同权限的对象对目标对象的访问，但在JavaScript并不容易实现保护代理，因为我们无法判断谁访问了某个对象。而虚拟代理是最常用的一种代理模式，本章主要讨论的也是虚拟代理。

当然上面只是一个虚拟的例子，我们无需在此投入过多近精力，接下来我们看另外一个真实的示例。

## 6.3 虚拟代理实现图片预加载

在Web开发中，图片预加载是一种常用的技术，如果直接给某个img 标签节点设置src 属性， 由于图片过大或者网络不佳，图片的位置往往有段时间会是一片空白。常见的做法是先用一张loading图片占位，然后用异步的方式加载图片，等图片加载好了再把它填充到img 节点里，这种场景就很适合使用虚拟代理。

下面我们来实现这个虚拟代理，首先创建一个普通的本体对象，这个对象负责往页面中创建一个img 标签，并且提供一个对外的setSrc 接口，外界调用这个接口，便可以给该img 标签设置src 属性：

```
var myImage = (function(){
    var imgNode = document.createElement( 'img' );
    document.body.appendChild( imgNode );

    return {
        setSrc: function( src ){
            imgNode.src = src;
        }
    }
})();

myImage.setSrc( 'http:// imgcache.qq.com/music/photo/k/000GGDys0yA0Nk.jp
```

我们把网速调至5KB/s，然后通过MyImage.setSrc 给该img 节点设置src ，可以看到，在图片被加载好之前，页面中有一段长长的空白时间。

现在开始引入代理对象proxyImage ，通过这个代理对象，在图片被真正加载好之前，页面中将出现一张占位的菊花图loading.gif, 来提示用户图片正在加载。代码如下：

```
var myImage = (function(){
    var imgNode = document.createElement( 'img' );
    document.body.appendChild( imgNode );
```

```
    return {
        setSrc: function( src ){
            imgNode.src = src;
        }
    }
})();

var proxyImage = (function(){
    var img = new Image;
    img.onload = function(){
        myImage.setSrc( this.src );
    }
    return {
        setSrc: function( src ){
            myImage.setSrc( 'file:// /C:/Users/svenzeng/Desktop/loading.
            img.src = src;
        }
    }
})();

proxyImage.setSrc( 'http:// imgcache.qq.com/music/photo/k/000GGDys0yA0Nk
```

现在我们通过proxyImage 间接地访问MyImage 。proxyImage 控制了客户对MyImage 的访问，并且在此过程中加入一些额外的操作，比如在真正的图片加载好之前，先把img 节点的src 设置为一张本地的loading图片。

## 6.4 代理的意义

也许读者会有疑问，不过是实现一个小小的图片预加载功能，即使不需要引入任何模式也能办到，那么引入代理模式的好处究竟在哪里呢？下面我们先抛开代理，编写一个更常见的图片预加载函数。

不用代理的预加载图片函数实现如下：

```
var MyImage = (function(){
    var imgNode = document.createElement( 'img' );
    document.body.appendChild( imgNode );
    var img = new Image;

    img.onload = function(){
        imgNode.src = img.src;
    };

    return {
        setSrc: function( src ){
            imgNode.src = 'file:// /C:/Users/svenzeng/Desktop/loading.gi
            img.src = src;
        }
    }
})();

MyImage.setSrc( 'http:// imgcache.qq.com/music/photo/k/000GGDys0yA0Nk.jp
```

为了说明代理的意义，下面我们引入一个面向对象设计的原则——单一职责原则。

单一职责原则指的是，就一个类（通常也包括对象和函数等）而言，应该仅有一个引起它变化的原因。如果一个对象承担了多项职责，就意味着这个对象将变得巨大，引起它变化的原因可能会有多个。面向对象设计鼓励将行为分布到细粒度的对象之中，如果一个对象承担的职责过多，等于把这些职责耦合到了一起，这种耦合会导致脆弱和低内聚的设计。当变化发生时，设计可能会遭到意外的破坏。

职责被定义为“引起变化的原因”。上段代码中的MyImage 对象除了负责给 img 节点设置src 外，还要负责预加载图片。我们在处理其中一个职责时，有可能因为其强耦合性影响另外一个职责的实现。

另外，在面向对象的程序设计中，大多数情况下，若违反其他任何原则，同时将违反开放—封闭原则。如果我们只是从网络上获取一些体积很小的图片，或者5年后的网速快到根本不再需要预加载，我们可能希望把预加载图片的这段代码从MyImage 对象里删掉。这时候就不得不改动MyImage 对象了。

实际上，我们需要的只是给img 节点设置src ，预加载图片只是一个锦上添花的功能。如果能把这个操作放在另一个对象里面，自然是一个非常好的方法。于是代理的作用在这里就体现出来了，代理负责预加载图片，预加载的操作完成之后，把请求重新交给本体MyImage 。

纵观整个程序，我们并没有改变或者增加MyImage 的接口，但是通过代理对象，实际上给系统添加了新的行为。这是符合开放—封闭原则的。给 img 节点设置src 和图片预加载这两个功能，被隔离在两个对象里，它们可以各自变化而不影响对方。何况就算有一天我们不再需要预加载，那么只需要改成请求本体而不是请求代理对象即可。

## 6.5 代理和本体接口的一致性

上一节说到，如果有一天我们不再需要预加载，那么就不再需要代理对象，可以选择直接请求本体。其中关键是代理对象和本体都对外提供了 `setSrc` 方法，在客户看来，代理对象和本体是一致的， 代理接手请求的过程对于用户来说是透明的，用户并不清楚代理和本体的区别，这样做有两个好处。

- 用户可以放心地请求代理，他只关心是否能得到想要的结果。
- 在任何使用本体的地方都可以替换成使用代理。

在Java等语言中，代理和本体都需要显式地实现同一个接口，一方面接口保证了它们会拥有同样的方法，另一方面，面向接口编程迎合依赖倒置原则，通过接口进行向上转型，从而避开编译器的类型检查，代理和本体将来可以被替换使用。

在JavaScript这种动态类型语言中，我们有时通过鸭子类型来检测代理和本体是否都实现了`setSrc` 方法，另外大多数时候甚至干脆不做检测，全部依赖程序员的自觉性，这对于程序的健壮性是有影响的。不过对于一门快速开发的脚本语言，这些影响还是在可以接受的范围内，而且我们也习惯了没有接口的世界。

另外值得一提的是，如果代理对象和本体对象都为一个函数（函数也是对象），函数必然都能被执行，则可以认为它们也具有一致的“接口”，代码如下：

```
var myImage = (function(){
    var imgNode = document.createElement( 'img' );
    document.body.appendChild( imgNode );

    return function( src ){
        imgNode.src = src;
    }
})();

var proxyImage = (function(){
    var img = new Image;

    img.onload = function(){
```

```
        myImage( this.src );
    }

    return function( src ){
        myImage( 'file:// /C:/Users/svenzeng/Desktop/loading.gif' );
        img.src = src;
    }
})();

proxyImage( 'http:// imgcache.qq.com/music// N/k/000GGDys0yA0Nk.jpg' );
```

## 6.6　虚拟代理合并HTTP请求

先想象这样一个场景：每周我们都要写一份工作周报，周报要交给总监批阅。总监手下管理着150个员工，如果我们每个人直接把周报发给总监，那总监可能要把一整周的时间都花在查看邮件上面。

现在我们把周报发给各自的组长，组长作为代理，把组内成员的周报合并提炼成一份后一次性地发给总监。这样一来，总监的邮箱便清净多了。

这个例子在程序世界里很容易引起共鸣，在Web开发中，也许最大的开销就是网络请求。假设我们在做一个文件同步的功能，当我们选中一个checkbox的时候，它对应的文件就会被同步到另外一台备用服务器上面，如图6-3所示。


图　6-3

我们先在页面中放置好这些checkbox节点：

```
<body>
    <input type="checkbox" id="1"></input>1
    <input type="checkbox" id="2"></input>2
    <input type="checkbox" id="3"></input>3
    <input type="checkbox" id="4"></input>4
    <input type="checkbox" id="5"></input>5
    <input type="checkbox" id="6"></input>6
```

```
    <input type="checkbox" id="7"></input>7
    <input type="checkbox" id="8"></input>8
    <input type="checkbox" id="9"></input>9
</body>
```

接下来，给这些checkbox绑定点击事件，并且在点击的同时往另一台服务器同步文件：

```
var synchronousFile = function( id ){
     console.log( '开始同步文件，id为：' + id );
};

var checkbox = document.getElementsByTagName( 'input' );

for ( var i = 0, c; c = checkbox[ i++ ]; ){
    c.onclick = function(){
        if ( this.checked === true ){
            synchronousFile( this.id );
        }
    }
};
```

当我们选中3个checkbox的时候，依次往服务器发送了3次同步文件的请求。而点击一个checkbox并不是很复杂的操作，作为APM250+的资深Dota玩家，我有把握一秒钟之内点中4个checkbox。可以预见，如此频繁的网络请求将会带来相当大的开销。

解决方案是，我们可以通过一个代理函数`proxySynchronousFile`来收集一段时间之内的请求，最后一次性发送给服务器。比如我们等待2秒之后才把这2秒之内需要同步的文件ID打包发给服务器，如果不是对实时性要求非常高的系统，2秒的延迟不会带来太大副作用，却能大大减轻服务器的压力。代码如下：

```
var synchronousFile = function( id ){
    console.log( '开始同步文件，id为：' + id );
};
```

```
var proxySynchronousFile = (function(){
    var cache = [],    // 保存一段时间内需要同步的ID
        timer;    // 定时器

     return function( id ){
        cache.push( id );
        if ( timer ){    // 保证不会覆盖已经启动的定时器
            return;
        }

        timer = setTimeout(function(){
            synchronousFile( cache.join( ',' ) );    // 2秒后向本体发送需要
            clearTimeout( timer );    // 清空定时器
            timer = null;
            cache.length = 0; // 清空ID集合
        }, 2000 );
    }
})();

var checkbox = document.getElementsByTagName( 'input' );

for ( var i = 0, c; c = checkbox[ i++ ]; ){
    c.onclick = function(){
        if ( this.checked === true ){
            proxySynchronousFile( this.id );
        }
    }
};
```

## 6.7　虚拟代理在惰性加载中的应用

我曾经写过一个mini控制台的开源项目miniConsole.js，这个控制台可以帮助开发者在IE浏览器以及移动端浏览器上进行一些简单的调试工作。调用方式很简单：

```
miniConsole.log(1);
```

这句话会在页面中创建一个div，并且把log显示在div里面，如图6-4所示。

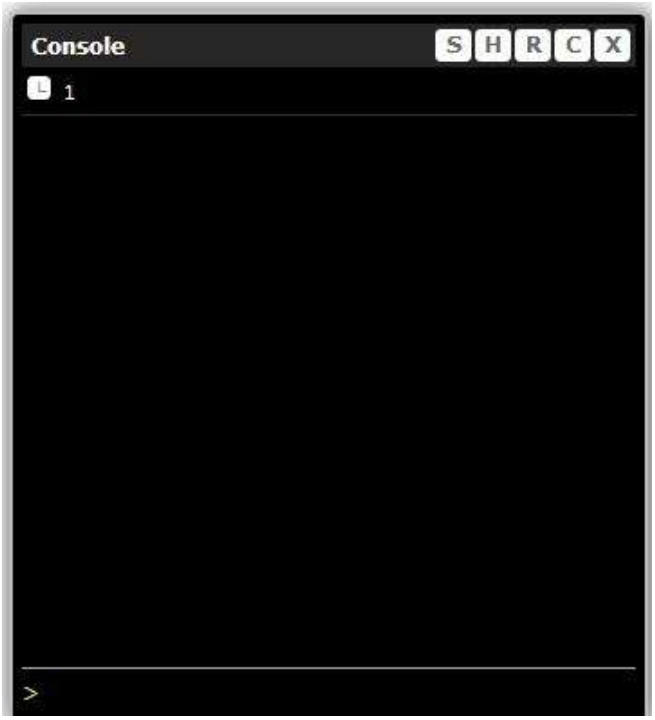



图　6-4

miniConsole.js的代码量大概有1000行左右，也许我们并不想一开始就加载这么大的JS文件， 因为也许并不是每个用户都需要打印log。我们希望在有必要的时候才开始加载它，比如当用户按下F2来主动唤出控制台的时候。

在miniConsole.js加载之前，为了能够让用户正常地使用里面的API，通常

我们的解决方案是用一个占位的miniConsole 代理对象来给用户提前使用，这个代理对象提供给用户的接口，跟实际的miniConsole 是一样的。

用户使用这个代理对象来打印log的时候，并不会真正在控制台内打印日志，更不会在页面中创建任何DOM节点。即使我们想这样做也无能为力，因为真正的miniConsole.js还没有被加载。

于是，我们可以把打印log的请求都包裹在一个函数里面，这个包装了请求的函数就相当于其他语言中命令模式中的Command 对象。随后这些函数将全部被放到缓存队列中，这些逻辑都是在miniConsole 代理对象中完成实现的。等用户按下F2唤出控制台的时候，才开始加载真正的miniConsole.js的代码，加载完成之后将遍历miniConsole 代理对象中的缓存函数队列，同时依次执行它们。

当然，请求的到底是什么对用户来说是不透明的，用户并不清楚它请求的是代理对象，所以他可以在任何时候放心地使用miniConsole 对象。

未加载真正的miniConsole.js之前的代码如下：

```
var cache = [];

var miniConsole = {
    log: function(){
        var args = arguments;
        cache.push( function(){
            return miniConsole.log.apply( miniConsole, args );
        });
    }
};

miniConsole.log(1);
```

当用户按下F2时，开始加载真正的miniConsole.js，代码如下：

```
var handler = function( ev ){
    if ( ev.keyCode === 113 ){
        var script = document.createElement( 'script' );
```

```
        script.onload = function(){
            for ( var i = 0, fn; fn = cache[ i++ ]; ){
                fn();
            }
        };
       script.src = 'miniConsole.js';
       document.getElementsByTagName( 'head' )[0].appendChild( script );
    }
};

document.body.addEventListener( 'keydown', handler, false );

// miniConsole.js代码:

miniConsole = {
    log: function(){
        // 真正代码略
        console.log( Array.prototype.join.call( arguments ) );
    }
};
```

虽然我们没有给出miniConsole.js的真正代码，但这不影响我们理解其中的逻辑。当然这里还要注意一个问题，就是我们要保证在F2被重复按下的时候，miniConsole.js只被加载一次。另外我们整理一下`miniConsole`代理对象的代码，使它成为一个标准的虚拟代理对象，代码如下：

```
var miniConsole = (function(){
    var cache = [];
    var handler = function( ev ){
        if ( ev.keyCode === 113 ){
            var script = document.createElement( 'script' );
            script.onload = function(){
                for ( var i = 0, fn; fn = cache[ i++ ]; ){
                    fn();
                }
            };
            script.src = 'miniConsole.js';
            document.getElementsByTagName( 'head' )[0].appendChild( scri
            document.body.removeEventListener( 'keydown', handler );// }
        }
    };

    document.body.addEventListener( 'keydown', handler, false );
```

```
    return {
        log: function(){
            var args = arguments;
                cache.push( function(){
                    return miniConsole.log.apply( miniConsole, args );
                });
          }
    }
})();

miniConsole.log( 11 );      // 开始打印log

// miniConsole.js代码

miniConsole = {
    log: function(){
        // 真正代码略
        console.log( Array.prototype.join.call( arguments ) );
    }
};
```

## 6.8 缓存代理

缓存代理可以为一些开销大的运算结果提供暂时的存储，在下次运算时，如果传递进来的参数跟之前一致，则可以直接返回前面存储的运算结果。

### 6.8.1 缓存代理的例子——计算乘积

为了节省示例代码，以及让读者把注意力集中在代理模式上面，这里编写一个简单的求乘积的程序，请读者自行把它脑补为复杂的计算。

先创建一个用于求乘积的函数：

```
var mult = function(){
    console.log( '开始计算乘积' );
    var a = 1;
    for ( var i = 0, l = arguments.length; i < l; i++ ){
        a = a * arguments[i];
    }
    return a;
};

mult( 2, 3 );     // 输出：6
mult( 2, 3, 4 );     // 输出：24
```

现在加入缓存代理函数：

```
var proxyMult = (function(){
    var cache = {};
    return function(){
        var args = Array.prototype.join.call( arguments, ',' );
        if ( args in cache ){
            return cache[ args ];
        }
        return cache[ args ] = mult.apply( this, arguments );
    }
})();

 proxyMult( 1, 2, 3, 4 );     // 输出：24
 proxyMult( 1, 2, 3, 4 );     // 输出：24
```

当我们第二次调用proxyMult( 1, 2, 3, 4 )的时候，本体mult 函数并没有被计算，proxyMult 直接返回了之前缓存好的计算结果。

通过增加缓存代理的方式，mult 函数可以继续专注于自身的职责——计算乘积，缓存的功能是由代理对象实现的。

### 6.8.2 缓存代理用于ajax异步请求数据

我们在常常在项目中遇到分页的需求，同一页的数据理论上只需要去后台拉取一次，这些已经拉取到的数据在某个地方被缓存之后，下次再请求同一页的时候，便可以直接使用之前的数据。

显然这里也可以引入缓存代理，实现方式跟计算乘积的例子差不多，唯一不同的是，请求数据是个异步的操作，我们无法直接把计算结果放到代理对象的缓存中，而是要通过回调的方式。具体代码不再赘述，读者可以自行实现。

## 6.9 用高阶函数动态创建代理

通过传入高阶函数这种更加灵活的方式，可以为各种计算方法创建缓存代理。现在这些计算方法被当作参数传入一个专门用于创建缓存代理的工厂中， 这样一来，我们就可以为乘法、加法、减法等创建缓存代理，代码如下：

```
/**************** 计算乘积 *****************/
var mult = function(){
    var a = 1;
    for ( var i = 0, l = arguments.length; i < l; i++ ){
        a = a * arguments[i];
    }
    return a;
};

/**************** 计算加和 *****************/
var plus = function(){
    var a = 0;
    for ( var i = 0, l = arguments.length; i < l; i++ ){
        a = a + arguments[i];
    }
    return a;
};

/**************** 创建缓存代理的工厂 *****************/
var createProxyFactory = function( fn ){
    var cache = {};
    return function(){
        var args = Array.prototype.join.call( arguments, ',' );
        if ( args in cache ){
            return cache[ args ];
        }
        return  cache[ args ] = fn.apply( this, arguments );
    }
};

var proxyMult = createProxyFactory( mult ),
proxyPlus = createProxyFactory( plus );

alert ( proxyMult( 1, 2, 3, 4 ) );    // 输出：24
alert ( proxyMult( 1, 2, 3, 4 ) );    // 输出：24
alert ( proxyPlus( 1, 2, 3, 4 ) );    // 输出：10
alert ( proxyPlus( 1, 2, 3, 4 ) );    // 输出：10
```

## 6.10　其他代理模式

代理模式的变体种类非常多，限于篇幅及其在JavaScript中的适用性，本章只简约介绍一下这些代理，就不一一详细展开说明了。

- 防火墙代理：控制网络资源的访问，保护主题不让“坏人”接近。
- 远程代理：为一个对象在不同的地址空间提供局部代表，在Java中，远程代理可以是另一个虚拟机中的对象。
- 保护代理：用于对象应该有不同访问权限的情况。
- 智能引用代理：取代了简单的指针，它在访问对象时执行一些附加操作，比如计算一个对象被引用的次数。
- 写时复制代理：通常用于复制一个庞大对象的情况。写时复制代理延迟了复制的过程，当对象被真正修改时，才对它进行复制操作。写时复制代理是虚拟代理的一种变体，DLL（操作系统中的动态链接库）是其典型运用场景。