第 12 章　享元模式

享元（flyweight）模式是一种用于性能优化的模式，“fly”在这里是苍蝇的意思，意为蝇量级。享元模式的核心是运用共享技术来有效支持大量细粒度的对象。

如果系统中因为创建了大量类似的对象而导致内存占用过高，享元模式就非常有用了。在JavaScript中，浏览器特别是移动端的浏览器分配的内存并不算多，如何节省内存就成了一件非常有意义的事情。

享元模式的概念初听起来并不太好理解，所以在深入讲解之前，我们先看一个例子。

12.1 初识享元模式

假设有个内衣工厂，目前的产品有50种男式内衣和50种女士内衣，为了推销产品，工厂决定生产一些塑料模特来穿上他们的内衣拍成广告照片。 正常情况下需要50个男模特和50个女模特，然后让他们每人分别穿上一件内衣来拍照。不使用享元模式的情况下，在程序里也许会这样写：

```
var Model = function( sex, underwear){
    this.sex = sex;
    this.underwear= underwear;
};

Model.prototype.takePhoto = function(){
    console.log( 'sex= ' + this.sex + ' underwear=' + this.underwear);
};

for ( var i = 1; i <= 50; i++ ){
    var maleModel = new Model( 'male', 'underwear' + i );
    maleModel.takePhoto();
};

for ( var j = 1; j <= 50; j++ ){
    var femaleModel= new Model( 'female', 'underwear' + j );
    femaleModel.takePhoto();
};
```

要得到一张照片，每次都需要传入sex 和underwear 参数，如上所述，现在一共有50种男内衣和50种女内衣，所以一共会产生100个对象。如果将来生产了10000种内衣，那这个程序可能会因为存在如此多的对象已经提前崩溃。

下面我们来考虑一下如何优化这个场景。虽然有100种内衣，但很显然并不需要50个男模特和50个女模特。其实男模特和女模特各自有一个就足够了，他们可以分别穿上不同的内衣来拍照。

现在来改写一下代码，既然只需要区别男女模特，那我们先把underwear 参数从构造函数中移除，构造函数只接收sex 参数：

```
var Model = function( sex ){
    this.sex = sex;
};

Model.prototype.takePhoto = function(){
    console.log( 'sex= ' + this.sex + ' underwear=' + this.underwear);
};
```

分别创建一个男模特对象和一个女模特对象：

```
var maleModel = new Model( 'male' ),
    femaleModel = new Model( 'female' );
```

给男模特依次穿上所有的男装，并进行拍照：

```
for ( var i = 1; i <= 50; i++ ){
    maleModel.underwear = 'underwear' + i;
    maleModel.takePhoto();
};
```

同样，给女模特依次穿上所有的女装，并进行拍照：

```
for ( var j = 1; j <= 50; j++ ){
    femaleModel.underwear = 'underwear' + j;
    femaleModel.takePhoto();
};
```

可以看到，改进之后的代码，只需要两个对象便完成了同样的功能。

12.2　内部状态与外部状态

12.1节的这个例子便是享元模式的雏形，享元模式要求将对象的属性划分为内部状态与外部状态（状态在这里通常指属性）。享元模式的目标是尽量减少共享对象的数量，关于如何划分内部状态和外部状态，下面的几条经验提供了一些指引。

- 内部状态存储于对象内部。
- 内部状态可以被一些对象共享。
- 内部状态独立于具体的场景，通常不会改变。
- 外部状态取决于具体的场景，并根据场景而变化，外部状态不能被共享。

这样一来，我们便可以把所有内部状态相同的对象都指定为同一个共享的对象。而外部状态可以从对象身上剥离出来，并储存在外部。

剥离了外部状态的对象成为共享对象，外部状态在必要时被传入共享对象来组装成一个完整的对象。虽然组装外部状态成为一个完整对象的过程需要花费一定的时间，但却可以大大减少系统中的对象数量，相比之下，这点时间或许是微不足道的。因此，享元模式是一种用时间换空间的优化模式。

在上面的例子中，性别是内部状态，内衣是外部状态，通过区分这两种状态，大大减少了系统中的对象数量。通常来讲，内部状态有多少种组合，系统中便最多存在多少个对象，因为性别通常只有男女两种，所以该内衣厂商最多只需要2个对象。

使用享元模式的关键是如何区别内部状态和外部状态。可以被对象共享的属性通常被划分为内部状态，如同不管什么样式的衣服，都可以按照性别不同，穿在同一个男模特或者女模特身上，模特的性别就可以作为内部状态储存在共享对象的内部。而外部状态取决于具体的场景，并根据场景而变化，就像例子中每件衣服都是不同的，它们不能被一些对象共享，因此只能被划分为外部状态。

12.3　享元模式的通用结构

12.1节的示例初步展示了享元模式的威力，但这还不是一个完整的享元模式，在这个例子中还存在以下两个问题。

- 我们通过构造函数显式new出了男女两个`model` 对象，在其他系统中，也许并不是一开始就需要所有的共享对象。
- 给`model` 对象手动设置了`underwear` 外部状态，在更复杂的系统中，这不是一个最好的方式，因为外部状态可能会相当复杂，它们与共享对象的联系会变得困难。

我们通过一个对象工厂来解决第一个问题，只有当某种共享对象被真正需要时，它才从工厂中被创建出来。对于第二个问题，可以用一个管理器来记录对象相关的外部状态，使这些外部状态通过某个钩子和共享对象联系起来。

12.4 文件上传的例子

在微云上传模块的开发中，我们曾经借助享元模式提升了程序的性能。下面我们就讲述这个例子。

12.4.1 对象爆炸

在微云上传模块的开发中，我曾经经历过对象爆炸的问题。微云的文件上传功能虽然可以选择依照队列，一个一个地排队上传，但也支持同时选择2000个文件。每一个文件都对应着一个JavaScript上传对象的创建，在第一版开发中，的确往程序里同时new了2000个upload 对象，结果可想而知，Chrome中还勉强能够支撑，IE下直接进入假死状态。

微云支持好几种上传方式，比如浏览器插件、Flash和表单上传等，为了简化例子，我们先假设只有插件和Flash这两种。不论是插件上传，还是Flash上传，原理都是一样的，当用户选择了文件之后，插件和Flash都会通知调用Window下的一个全局JavaScript函数，它的名字是startUpload ，用户选择的文件列表被组合成一个数组files 塞进该函数的参数列表里，代码如下：

```
var id = 0;

window.startUpload = function( uploadType, files ){    // uploadType区分
    for ( var i = 0, file; file = files[ i++ ]; ){
        var uploadObj = new Upload( uploadType, file.fileName, file.file
        uploadObj.init( id++ );    // 给upload对象设置一个唯一的id
    }
};
```

当用户选择完文件之后，startUpload 函数会遍历files 数组来创建对应的upload 对象。接下来定义Upload 构造函数，它接受3个参数，分别是插件类型、文件名和文件大小。这些信息都已经被插件组装在files 数组里返回，代码如下：

```
var Upload = function( uploadType, fileName, fileSize ){
    this.uploadType = uploadType;
    this.fileName = fileName;
    this.fileSize = fileSize;
    this.dom= null;
};

Upload.prototype.init = function( id ){
    var that = this;
    this.id = id;
    this.dom = document.createElement( 'div' );
    this.dom.innerHTML =
                '<span>文件名称:'+ this.fileName +', 文件大小: '+ this.file
                '<button class="delFile">删除</button>';

    this.dom.querySelector( '.delFile' ).onclick = function(){
        that.delFile();
    }
    document.body.appendChild( this.dom );
};
```

同样为了简化示例，我们暂且去掉了upload 对象的其他功能，只保留删除文件的功能，对应的方法是Upload.prototype.delFile 。该方法中有一个逻辑：当被删除的文件小于3000 KB时，该文件将被直接删除。否则页面中会弹出一个提示框，提示用户是否确认要删除该文件，代码如下：

```
Upload.prototype.delFile = function(){
    if ( this.fileSize < 3000 ){
        return this.dom.parentNode.removeChild( this.dom );
    }

    if ( window.confirm( '确定要删除该文件吗? ' + this.fileName ) ){
        return this.dom.parentNode.removeChild( this.dom );
    }
};
```

接下来分别创建3个插件上传对象和3个Flash上传对象：

```
startUpload( 'plugin', [
    {
        fileName: '1.txt',
        fileSize: 1000
    },
    {
        fileName: '2.html',
        fileSize: 3000
    },
    {
        fileName: '3.txt',
        fileSize: 5000
    }
]);

startUpload( 'flash', [
    {
        fileName: '4.txt',
        fileSize: 1000
    },
    {
        fileName: '5.html',
        fileSize: 3000
    },
    {
        fileName: '6.txt',
        fileSize: 5000
    }
]);
```

当点击删除最后一个文件时，可以看到弹出了是否确认删除的提示，如图12-1所示。

图　12-1

12.4.2 享元模式重构文件上传

上一节的代码是第一版的文件上传，在这段代码里有多少个需要上传的文件，就一共创建了多少个upload 对象，接下来我们用享元模式重构它。

首先，我们需要确认插件类型uploadType 是内部状态，那为什么单单uploadType 是内部状态呢？前面讲过，划分内部状态和外部状态的关键主要有以下几点。

- 内部状态储存于对象内部。
- 内部状态可以被一些对象共享。
- 内部状态独立于具体的场景，通常不会改变。
- 外部状态取决于具体的场景，并根据场景而变化，外部状态不能被共享。

在文件上传的例子里，upload 对象必须依赖uploadType 属性才能工作，这是因为插件上传、Flash上传、表单上传的实际工作原理有很大的区别，它们各自调用的接口也是完全不一样的，必须在对象创建之初就明确它是什么类型的插件，才可以在程序的运行过程中，让它们分别调用各自的start 、pause 、cancel 、del 等方法。

实际上在微云的真实代码中，虽然插件和Flash上传对象最终创建自一个大的工厂类，但它们实际上根据uploadType 值的不同，分别是来自于两个不同类的对象。（在目前的例子中，为了简化代码，我们把插件和Flash的构造函数合并成了一个。）

一旦明确了uploadType ，无论我们使用什么方式上传，这个上传对象都是可以被任何文件共用的。而fileName 和fileSize 是根据场景而变化的，每个文件的fileName 和fileSize 都不一样，fileName 和fileSize 没有办法被共享，它们只能被划分为外部状态。

12.4.3 剥离外部状态

明确了uploadType 作为内部状态之后，我们再把其他的外部状态从构造

函数中抽离出来， Upload 构造函数中只保留uploadType 参数：

```
var Upload = function( uploadType){
    this.uploadType = uploadType;
};
```

Upload.prototype.init 函数也不再需要，因为upload 对象初始化的工作被放在了upload- Manager.add 函数里面，接下来只需要定义Upload.prototype.del 函数即可：

```
Upload.prototype.delFile = function( id ){
    uploadManager.setExternalState( id, this );  // (1)

    if ( this.fileSize < 3000 ){
        return this.dom.parentNode.removeChild( this.dom );
    }

    if ( window.confirm( '确定要删除该文件吗? ' + this.fileName ) ){
        return this.dom.parentNode.removeChild( this.dom );
    }
};
```

在开始删除文件之前，需要读取文件的实际大小，而文件的实际大小被储存在外部管理器uploadManager 中，所以在这里需要通过 uploadManager.setExternalState 方法给共享对象设置正确的 fileSize ，上段代码中的(1)处表示把当前id 对应的对象的外部状态都组装到共享对象中。

12.4.4 工厂进行对象实例化

接下来定义一个工厂来创建upload 对象，如果某种内部状态对应的共享对象已经被创建过，那么直接返回这个对象，否则创建一个新的对象：

```
var UploadFactory = (function(){
```

```
    var createdFlyWeightObjs = {};

    return {
        create: function( uploadType){
            if ( createdFlyWeightObjs [ uploadType] ){
                return createdFlyWeightObjs [ uploadType];
            }

            return createdFlyWeightObjs [ uploadType] = new Upload( uplo
        }
    }
})();
```

12.4.5 管理器封装外部状态

现在我们来完善前面提到的uploadManager 对象，它负责向UploadFactory 提交创建对象的请求，并用一个uploadDatabase 对象保存所有upload 对象的外部状态，以便在程序运行过程中给upload 共享对象设置外部状态，代码如下：

```
var uploadManager = (function(){
    var uploadDatabase = {};

    return {
        add: function( id, uploadType, fileName, fileSize ){
            var flyWeightObj = UploadFactory.create( uploadType );

            var dom = document.createElement( 'div' );
            dom.innerHTML =
                    '<span>文件名称:'+ fileName +', 文件大小: '+ fileSize
                    '<button class="delFile">删除</button>';

            dom.querySelector( '.delFile' ).onclick = function(){
                flyWeightObj.delFile( id );
            }

            document.body.appendChild( dom );

            uploadDatabase[ id ] = {
                fileName: fileName,
                fileSize: fileSize,
                dom: dom
```

```
            };

            return flyWeightObj ;
        },
        setExternalState: function( id, flyWeightObj ){
            var uploadData = uploadDatabase[ id ];
            for ( var i in uploadData ){
                flyWeightObj[ i ] = uploadData[ i ];
            }
        }
    }
})();
```

然后是开始触发上传动作的startUpload 函数：

```
var id = 0;

window.startUpload = function( uploadType, files ){
   for ( var i = 0, file; file = files[ i++ ]; ){
        var uploadObj = uploadManager.add( ++id, uploadType, file.fileNa
   }
};
```

最后是测试时间，运行下面的代码后，可以发现运行结果跟用享元模式重构之前一致：

```
startUpload( 'plugin', [
    {
        fileName: '1.txt',
        fileSize: 1000
    },
    {
        fileName: '2.html',
        fileSize: 3000
    },
    {
        fileName: '3.txt',
        fileSize: 5000
    }
```

```
]);

startUpload( 'flash', [
    {
        fileName: '4.txt',
        fileSize: 1000
    },
    {
        fileName: '5.html',
        fileSize: 3000
    },
    {
        fileName: '6.txt',
        fileSize: 5000
    }
]);
```

享元模式重构之前的代码里一共创建了6个upload 对象，而通过享元模式重构之后，对象的数量减少为2，更幸运的是， 就算现在同时上传2000个文件，需要创建的upload 对象数量依然是2。

12.5 享元模式的适用性

享元模式是一种很好的性能优化方案，但它也会带来一些复杂性的问题，从前面两组代码的比较可以看到，使用了享元模式之后，我们需要分别多维护一个`factory`对象和一个`manager`对象，在大部分不必要使用享元模式的环境下，这些开销是可以避免的。

享元模式带来的好处很大程度上取决于如何使用以及何时使用，一般来说，以下情况发生时便可以使用享元模式。

- 一个程序中使用了大量的相似对象。
- 由于使用了大量对象，造成很大的内存开销。
- 对象的大多数状态都可以变为外部状态。
- 剥离出对象的外部状态之后，可以用相对较少的共享对象取代大量对象。

可以看到，文件上传的例子完全符合这四点。

12.6 再谈内部状态和外部状态

如果顺利的话，通过前面的例子我们已经了解了内部状态和外部状态的概念以及享元模式的工作原理。我们知道，实现享元模式的关键是把内部状态和外部状态分离开来。有多少种内部状态的组合，系统中便最多存在多少个共享对象，而外部状态储存在共享对象的外部，在必要时被传入共享对象来组装成一个完整的对象。现在来考虑两种极端的情况，即对象没有外部状态和没有内部状态的时候。

12.6.1 没有内部状态的享元

在文件上传的例子中，我们分别进行过插件调用和Flash调用，即`startUpload( 'plugin', [] )`和`startUpload( flash, [] )`，导致程序中创建了内部状态不同的两个共享对象。也许你会奇怪，在文件上传程序里，一般都会提前通过特性检测来选择一种上传方式，如果浏览器支持插件就用插件上传，如果不支持插件，就用Flash上传。那么，什么情况下既需要插件上传又需要Flash上传呢？

实际上这个需求是存在的，很多网盘都提供了极速上传（控件）与普通上传（Flash）两种模式，如果极速上传不好使（可能是没有安装控件或者控件损坏），用户还可以随时切换到普通上传模式，所以这里确实是需要同时存在两个不同的`upload` 共享对象。

但不是每个网站都必须做得如此复杂，很多小一些的网站就只支持单一的上传方式。假设我们是这个网站的开发者，不需要考虑极速上传与普通上传之间的切换，这意味着在之前的代码中作为内部状态的`uploadType` 属性是可以删除掉的。

在继续使用享元模式的前提下，构造函数`Upload` 就变成了无参数的形式：

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

其他属性如fileName、fileSize、dom依然可以作为外部状态保存在共享对象外部。在uploadType作为内部状态的时候，它可能为控件，也可能为Flash，所以当时最多可以组合出两个共享对象。而现在已经没有了内部状态，这意味着只需要唯一的一个共享对象。现在我们要改写创建享元对象的工厂，代码如下：

```
var UploadFactory = (function(){
    var uploadObj;
    return {
        create: function(){
            if ( uploadObj ){
                return uploadObj;
            }
            return uploadObj = new Upload();
        }
    }
})();
```

管理器部分的代码不需要改动，还是负责剥离和组装外部状态。可以看到，当对象没有内部状态的时候，生产共享对象的工厂实际上变成了一个单例工厂。虽然这时候的共享对象没有内部状态的区分，但还是有剥离外部状态的过程，我们依然倾向于称之为享元模式。

12.6.2 没有外部状态的享元

网上许多资料中，经常把Java或者C#的字符串看成享元，这种说法是否正确呢？我们看看下面这段Java代码，来分析一下：

```
// Java代码

public class Test {

    public static void main( String args[] ){
        String a1 = new String( "a" ).intern();
        String a2 = new String( "a" ).intern();
        System.out.println( a1 == a2 );    // true
    }
}
```

在这段Java代码里，分别new了两个字符串对象a1 和a2 。intern 是一种对象池技术， new String("a").intern() 的含义如下。

- 如果值为a 的字符串对象已经存在于对象池中，则返回这个对象的引用。
- 反之，将字符串a的对象添加进对象池，并返回这个对象的引用。

所以a1 == a2 的结果是true ，但这并不是使用了享元模式的结果，享元模式的关键是区别内部状态和外部状态。享元模式的过程是剥离外部状态，并把外部状态保存在其他地方，在合适的时刻再把外部状态组装进共享对象。这里并没有剥离外部状态的过程，a1 和a2 指向的完全就是同一个对象，所以如果没有外部状态的分离，即使这里使用了共享的技术，但并不是一个纯粹的享元模式。

12.7　对象池

我们在前面已经提到了Java中`String`的对象池，下面就来学习这种共享的技术。对象池维护一个装载空闲对象的池子，如果需要对象的时候，不是直接new，而是转从对象池里获取。如果对象池里没有空闲对象，则创建一个新的对象，当获取出的对象完成它的职责之后， 再进入池子等待被下次获取。

对象池的原理很好理解，比如我们组人手一本《JavaScript权威指南》，从节约的角度来讲，这并不是很划算，因为大部分时间这些书都被闲置在各自的书架上，所以我们一开始就只买一本，或者一起建立一个小型图书馆（对象池），需要看书的时候就从图书馆里借，看完了之后再把书还回图书馆。如果同时有三个人要看这本书，而现在图书馆里只有两本，那我们再马上去书店买一本放入图书馆。

对象池技术的应用非常广泛，HTTP连接池和数据库连接池都是其代表应用。在Web前端开发中，对象池使用最多的场景大概就是跟DOM有关的操作。很多空间和时间都消耗在了DOM节点上，如何避免频繁地创建和删除DOM节点就成了一个有意义的话题。

12.7.1　对象池实现

假设我们在开发一个地图应用， 地图上经常会出现一些标志地名的小气泡，我们叫它`toolTip`。如图12-2所示。

图　12-2

在搜索我家附近地图的时候，页面里出现了2个小气泡。当我再搜索附近的兰州拉面馆时，页面中出现了6个小气泡。按照对象池的思想，在第二次搜索开始之前，并不会把第一次创建的2个小气泡删除掉，而是把它们放进对象池。这样在第二次的搜索结果页面里，我们只需要再创建4个小气泡而不是6个，如图12-3所示。

图　12-3

先定义一个获取小气泡节点的工厂，作为对象池的数组成为私有属性被包含在工厂闭包里，这个工厂有两个暴露对外的方法，create 表示获取一个div 节点，recover 表示回收一个div 节点：

```
var toolTipFactory = (function(){
    var toolTipPool = [];    // toolTip对象池

    return {
        create: function(){
            if ( toolTipPool.length === 0 ){    // 如果对象池为空
                var div = document.createElement( 'div' );    // 创建一个
                document.body.appendChild( div );
                return div;
            }else{    // 如果对象池里不为空
                return toolTipPool.shift();  // 则从对象池中取出一个dom
            }
        },
        recover: function( tooltipDom ){
            return toolTipPool.push( tooltipDom );    // 对象池回收dom
        }
```

```
    }
})();
```

现在把时钟拨回进行第一次搜索的时刻，目前需要创建2个小气泡节点，为了方便回收，用一个数组ary 来记录它们：

```
var ary = [];

for ( var i = 0, str; str = [ 'A', 'B' ][ i++ ]; ){
    var toolTip = toolTipFactory.create();
    toolTip.innerHTML = str;
    ary.push( toolTip );
};
```

如果你愿意稍稍测试一下，可以看到页面中出现了innerHTML 分别为A和B的两个div 节点。

接下来假设地图需要开始重新绘制，在此之前要把这两个节点回收进对象池：

```
for ( var i = 0, toolTip; toolTip = ary[ i++ ]; ){
    toolTipFactory.recover( toolTip );
};
```

再创建6个小气泡：

```
for ( var i = 0, str; str = [ 'A', 'B', 'C', 'D', 'E', 'F' ][ i++ ]; ){
    var toolTip = toolTipFactory.create();
    toolTip.innerHTML = str;
};
```

现在再测试一番，页面中出现了内容分别为A、B、C、D、E、F的6个节点，上一次创建好的节点被共享给了下一次操作。对象池跟享元模式的思想有点相似，虽然innerHTML 的值A、B、C、D等也可以看成节点的外部状态，但在这里我们并没有主动分离内部状态和外部状态的过程。

12.7.2 通用对象池实现

我们还可以在对象池工厂里，把创建对象的具体过程封装起来，实现一个通用的对象池：

```
var objectPoolFactory = function( createObjFn ){
    var objectPool = [];

    return {
        create: function(){
            var obj = objectPool.length === 0 ?
                createObjFn.apply( this, arguments ) : objectPool.shift(

            return obj;
        },
        recover: function( obj ){
            objectPool.push( obj );
        }
    }
};
```

现在利用objectPoolFactory 来创建一个装载一些iframe 的对象池：

```
var iframeFactory = objectPoolFactory( function(){
    var iframe = document.createElement( 'iframe' );
       document.body.appendChild( iframe );

    iframe.onload = function(){
        iframe.onload = null;    // 防止iframe重复加载的bug
        iframeFactory.recover( iframe );    // iframe加载完成之后回收节点
    }
```

```
    return iframe;

});

var iframe1 = iframeFactory.create();
iframe1.src = 'http:// baidu.com';

var iframe2 = iframeFactory.create();
iframe2.src = 'http:// QQ.com';

setTimeout(function(){
     var iframe3 = iframeFactory.create();
    iframe3.src = 'http:// 163.com';
}, 3000 );
```

对象池是另外一种性能优化方案，它跟享元模式有一些相似之处，但没有分离内部状态和外部状态这个过程。本章用享元模式完成了一个文件上传的程序，其实也可以用对象池+事件委托来代替实现。