**介绍**

桥接模式（Bridge）将抽象部分与它的实现部分分离，使它们都可以独立地变化。

**正文**

桥接模式最常用在事件监控上，先看一段代码：

addEvent(element, 'click', getBeerById);  
function getBeerById(e) {  
var id = this.id;  
asyncRequest('GET', 'beer.uri?id=' + id, function(resp) {  
// Callback response.  
console.log('Requested Beer: ' + resp.responseText);  
});  
}

上述代码，有个问题就是getBeerById必须要有浏览器的上下文才能使用，因为其内部使用了this.id这个属性，如果没用上下文，那就歇菜了。所以说一般稍微有经验的程序员都会将程序改造成如下形式：

function getBeerById(id, callback) {  
// 通过ID发送请求，然后返回数据  
asyncRequest('GET', 'beer.uri?id=' + id, function(resp) {  
// callback response  
callback(resp.responseText);  
});  
}

实用多了，对吧？首先ID可以随意传入，而且还提供了一个callback函数用于自定义处理函数。但是这个和桥接有什么关系呢？这就是下段代码所要体现的了：

addEvent(element, 'click', getBeerByIdBridge);  
　　function getBeerByIdBridge (e) {  
　　　　getBeerById(this.id, function(beer) {  
　　　　　　console.log('Requested Beer: '+beer);  
　　});  
}

这里的getBeerByIdBridge就是我们定义的桥，用于将抽象的click事件和getBeerById连接起来，同时将事件源的ID，以及自定义的call函数（console.log输出）作为参数传入到getBeerById函数里。

这个例子看起来有些简单，我们再来一个复杂点的实战例子。

**实战XHR连接队列**

我们要构建一个队列，队列里存放了很多ajax请求，使用队列（queue）主要是因为要确保先加入的请求先被处理。任何时候，我们可以暂停请求、删除请求、重试请求以及支持对各个请求的订阅事件。

**基础核心函数**

在正式开始之前，我们先定义一下核心的几个封装函数，首先第一个是异步请求的函数封装：

var asyncRequest = (function () {  
 function handleReadyState(o, callback) {  
 var poll = window.setInterval(  
 function () {  
 if (o && o.readyState == 4) {  
 window.clearInterval(poll);  
 if (callback) {  
 callback(o);  
 }  
 }  
 },  
 50  
 );  
 }  
  
 var getXHR = function () {  
 var http;  
 try {  
 http = new XMLHttpRequest;  
 getXHR = function () {  
 return new XMLHttpRequest;  
 };  
 }  
  
 catch (e) {  
 var msxml = [  
 'MSXML2.XMLHTTP.3.0',  
 'MSXML2.XMLHTTP',  
 'Microsoft.XMLHTTP'  
 ];  
  
 for (var i = 0, len = msxml.length; i < len; ++i) {  
 try {  
 http = new ActiveXObject(msxml[i]);  
 getXHR = function () {  
 return new ActiveXObject(msxml[i]);  
 };  
 break;  
 }  
 catch (e) { }  
 }  
 }  
 return http;  
 };  
  
 return function (method, uri, callback, postData) {  
 var http = getXHR();  
 http.open(method, uri, true);  
 handleReadyState(http, callback);  
 http.send(postData || null);  
 return http;  
 };  
})();

上述封装的自执行函数是一个通用的Ajax请求函数，相信属性Ajax的人都能看懂了。

接下来我们定义一个通用的添加方法（函数）的方法：

Function.prototype.method = function (name, fn) {  
 this.prototype[name] = fn;  
 return this;  
};

最后再添加关于数组的2个方法，一个用于遍历，一个用于筛选：

if (!Array.prototype.forEach) {  
 Array.method('forEach', function (fn, thisObj) {  
 var scope = thisObj || window;  
 for (var i = 0, len = this.length; i < len; ++i) {  
 fn.call(scope, this[i], i, this);  
 }  
 });  
}  
  
if (!Array.prototype.filter) {  
 Array.method('filter', function (fn, thisObj) {  
 var scope = thisObj || window;  
 var a = [];  
 for (var i = 0, len = this.length; i < len; ++i) {  
 if (!fn.call(scope, this[i], i, this)) {  
 continue;  
 }  
 a.push(this[i]);  
 }  
 return a;  
 });  
}

因为有的新型浏览器已经支持了这两种功能（或者有些类库已经支持了），所以要先判断，如果已经支持的话，就不再处理了。

**观察者系统**

观察者在队列里的事件过程中扮演着重要的角色，可以队列处理时（成功、失败、挂起）订阅事件：

window.DED = window.DED || {};  
DED.util = DED.util || {};  
DED.util.Observer = function () {  
 this.fns = [];  
}  
  
DED.util.Observer.prototype = {  
 subscribe: function (fn) {  
 this.fns.push(fn);  
 },  
  
 unsubscribe: function (fn) {  
 this.fns = this.fns.filter(  
 function (el) {  
 if (el !== fn) {  
 return el;  
 }  
 }  
 );  
 },  
 fire: function (o) {  
 this.fns.forEach(  
 function (el) {  
 el(o);  
 }  
 );  
 }  
};

**队列主要实现代码**

首先订阅了队列的主要属性和事件委托：

DED.Queue = function () {  
 // 包含请求的队列.  
 this.queue = [];  
 // 使用Observable对象在3个不同的状态上，以便可以随时订阅事件  
 this.onComplete = new DED.util.Observer;  
 this.onFailure = new DED.util.Observer;  
 this.onFlush = new DED.util.Observer;  
  
 // 核心属性，可以在外部调用的时候进行设置  
 this.retryCount = 3;  
 this.currentRetry = 0;  
 this.paused = false;  
 this.timeout = 5000;  
 this.conn = {};  
 this.timer = {};  
};

然后通过DED.Queue.method的链式调用，则队列上添加了很多可用的方法：

DED.Queue.  
 method('flush', function () {  
 // flush方法  
 if (!this.queue.length > 0) {  
 return;  
 }  
  
 if (this.paused) {  
 this.paused = false;  
 return;  
 }  
  
 var that = this;  
 this.currentRetry++;  
 var abort = function () {  
 that.conn.abort();  
 if (that.currentRetry == that.retryCount) {  
 that.onFailure.fire();  
 that.currentRetry = 0;  
 } else {  
 that.flush();  
 }  
 };  
  
 this.timer = window.setTimeout(abort, this.timeout);  
 var callback = function (o) {  
 window.clearTimeout(that.timer);  
 that.currentRetry = 0;  
 that.queue.shift();  
 that.onFlush.fire(o.responseText);  
 if (that.queue.length == 0) {  
 that.onComplete.fire();  
 return;  
 }  
  
 // recursive call to flush  
 that.flush();  
  
 };  
  
 this.conn = asyncRequest(  
 this.queue[0]['method'],  
 this.queue[0]['uri'],  
 callback,  
 this.queue[0]['params']  
 );  
 }).  
 method('setRetryCount', function (count) {  
 this.retryCount = count;  
 }).  
 method('setTimeout', function (time) {  
 this.timeout = time;  
 }).  
 method('add', function (o) {  
 this.queue.push(o);  
 }).  
 method('pause', function () {  
 this.paused = true;  
 }).  
 method('dequeue', function () {  
 this.queue.pop();  
 }).  
 method('clear', function () {  
 this.queue = [];  
 });

代码看起来很多，折叠以后就可以发现，其实就是在队列上定义了flush, setRetryCount, setTimeout, add, pause, dequeue, 和clear方法。

**简单调用**

var q = new DED.Queue;  
// 设置重试次数高一点，以便应付慢的连接  
q.setRetryCount(5);  
// 设置timeout时间  
q.setTimeout(1000);  
// 添加2个请求.  
q.add({  
 method: 'GET',  
 uri: '/path/to/file.php?ajax=true'  
});  
  
q.add({  
 method: 'GET',  
 uri: '/path/to/file.php?ajax=true&woe=me'  
});  
  
// flush队列  
q.flush();  
// 暂停队列，剩余的保存  
q.pause();  
// 清空.  
q.clear();  
// 添加2个请求.  
q.add({  
 method: 'GET',  
 uri: '/path/to/file.php?ajax=true'  
});  
  
q.add({  
 method: 'GET',  
 uri: '/path/to/file.php?ajax=true&woe=me'  
});  
  
// 从队列里删除最后一个请求.  
q.dequeue();  
// 再次Flush  
q.flush();

**桥接呢？**

上面的调用代码里并没有桥接，那桥呢？看一下下面的完整示例，就可以发现处处都有桥哦：

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"  
"http://www.w3.org/TR/html4/strict.dtd">  
<html>  
<head>  
 <meta http-equiv="Content-type" content="text/html; charset=utf-8">  
 <title>Ajax Connection Queue</title>  
 <script src="utils.js"></script>  
 <script src="queue.js"></script>  
 <script type="text/javascript">  
 addEvent(window, 'load', function () {  
 // 实现.  
var q = new DED.Queue;  
 q.setRetryCount(5);  
 q.setTimeout(3000);  
 var items = $('items');  
 var results = $('results');  
 var queue = $('queue-items');  
 // 在客户端保存跟踪自己的请求  
var requests = [];  
 // 每个请求flush以后，订阅特殊的处理步骤  
 q.onFlush.subscribe(function (data) {  
 results.innerHTML = data;  
 requests.shift();  
 queue.innerHTML = requests.toString();  
 });  
 // 订阅时间处理步骤  
 q.onFailure.subscribe(function () {  
 results.innerHTML += ' <span style="color:red;">Connection Error!</span>';  
 });  
 // 订阅全部成功的处理步骤x  
 q.onComplete.subscribe(function () {  
 results.innerHTML += ' <span style="color:green;">Completed!</span>';  
 });  
 var actionDispatcher = function (element) {  
 switch (element) {  
 case 'flush':  
 q.flush();  
 break;  
 case 'dequeue':  
 q.dequeue();  
 requests.pop();  
 queue.innerHTML = requests.toString();  
 break;  
 case 'pause':  
 q.pause();  
 break;  
 case 'clear':  
 q.clear();  
 requests = [];  
 queue.innerHTML = '';  
 break;  
 }  
 };  
 var addRequest = function (request) {  
 var data = request.split('-')[1];  
 q.add({  
 method: 'GET',  
 uri: 'bridge-connection-queue.php?ajax=true&s=' + data,  
 params: null  
 });  
 requests.push(data);  
 queue.innerHTML = requests.toString();  
 };  
 addEvent(items, 'click', function (e) {  
 var e = e || window.event;  
 var src = e.target || e.srcElement;  
 try {  
 e.preventDefault();  
 }  
 catch (ex) {  
 e.returnValue = false;  
 }  
 actionDispatcher(src.id);  
 });  
 var adders = $('adders');  
 addEvent(adders, 'click', function (e) {  
 var e = e || window.event;  
 var src = e.target || e.srcElement;  
 try {  
 e.preventDefault();  
 }  
 catch (ex) {  
 e.returnValue = false;  
 }  
 addRequest(src.id);  
 });  
 });  
 </script>  
 <style type="text/css" media="screen">  
 body  
 {  
 font: 100% georgia,times,serif;  
 }  
 h1, h2  
 {  
 font-weight: normal;  
 }  
 #queue-items  
 {  
 height: 1.5em;  
 }  
 #add-stuff  
 {  
 padding: .5em;  
 background: #ddd;  
 border: 1px solid #bbb;  
 }  
 #results-area  
 {  
 padding: .5em;  
 border: 1px solid #bbb;  
 }  
 </style>  
</head>  
<body id="example">  
 <div id="doc">  
 <h1>  
 异步联接请求</h1>  
 <div id="queue-items">  
 </div>  
 <div id="add-stuff">  
 <h2>向队列里添加新请求</h2>  
 <ul id="adders">  
 <li><a href="#" id="action-01">添加 "01" 到队列</a></li>  
 <li><a href="#" id="action-02">添加 "02" 到队列</a></li>  
 <li><a href="#" id="action-03">添加 "03" 到队列</a></li>  
 </ul>  
 </div>  
 <h2>队列控制</h2>  
 <ul id='items'>  
 <li><a href="#" id="flush">Flush</a></li>  
 <li><a href="#" id="dequeue">出列Dequeue</a></li>  
 <li><a href="#" id="pause">暂停Pause</a></li>  
 <li><a href="#" id="clear">清空Clear</a></li>  
 </ul>  
 <div id="results-area">  
 <h2>  
 结果:  
 </h2>  
 <div id="results">  
 </div>  
 </div>  
 </div>  
</body>  
</html>

在这个示例里，你可以做flush队列，暂停队列，删除队列里的请求，清空队列等各种动作，同时相信大家也体会到了桥接的威力了。

**总结**

桥接模式的优点也很明显，我们只列举主要几个优点：

1. 分离接口和实现部分，一个实现未必不变地绑定在一个接口上，抽象类（函数）的实现可以在运行时刻进行配置，一个对象甚至可以在运行时刻改变它的实现，同将抽象和实现也进行了充分的解耦，也有利于分层，从而产生更好的结构化系统。
2. 提高可扩充性
3. 实现细节对客户透明，可以对客户隐藏实现细节。

同时桥接模式也有自己的缺点：

大量的类将导致开发成本的增加，同时在性能方面可能也会有所减少。