**介绍**

本篇我们介绍的一些模式称为初始化模式和性能模式，主要是用在初始化以及提高性能方面，一些模式之前已经提到过，这里只是做一下总结。

**立即执行的函数**

在本系列第4篇的《[立即调用的函数表达式](http://www.cnblogs.com/TomXu/archive/2011/12/31/2289423.html" \t "_blank)》中，我们已经对类似的函数进行过详细的描述，这里我们只是再举两个简单的例子做一下总结。

// 声明完函数以后，立即执行该函数

(function () {

console.log('watch out!');

} ());

//这种方式声明的函数，也可以立即执行

!function () {

console.log('watch out!');

} ();

// 如下方式也都可以哦

~function () { /\* code \*/ } ();

-function () { /\* code \*/ } ();

+function () { /\* code \*/ } ();

**立即执行的对象初始化**

该模式的意思是指在声明一个对象（而非函数）的时候，立即执行对象里的某一个方法来进行初始化工作，通常该模式可以用在一次性执行的代码上。

({

// 这里你可以定义常量，设置其它值

maxwidth: 600,

maxheight: 400,

// 当然也可以定义utility方法

gimmeMax: function () {

return this.maxwidth + "x" + this.maxheight;

},

// 初始化

init: function () {

console.log(this.gimmeMax());

// 更多代码...

}

}).init(); // 这样就开始初始化咯

**分支初始化**

分支初始化是指在初始化的时候，根据不同的条件（场景）初始化不同的代码，也就是所谓的条件语句赋值。之前我们在做事件处理的时候，通常使用类似下面的代码：

var utils = {

addListener: function (el, type, fn) {

if (typeof window.addEventListener === 'function') {

el.addEventListener(type, fn, false);

} else if (typeof document.attachEvent !== 'undefined') {

el.attachEvent('on' + type, fn);

} else {

el['on' + type] = fn;

}

},

removeListener: function (el, type, fn) {

}

};

我们来改进一下，首先我们要定义两个接口，一个用来add事件句柄，一个用来remove事件句柄，代码如下：

var utils = {

addListener: null,

removeListener: null

};

实现代码如下：

if (typeof window.addEventListener === 'function') {

utils.addListener = function (el, type, fn) {

el.addEventListener(type, fn, false);

};

} else if (typeof document.attachEvent !== 'undefined') { // IE

utils.addListener = function (el, type, fn) {

el.attachEvent('on' + type, fn);

};

utils.removeListener = function (el, type, fn) {

el.detachEvent('on' + type, fn);

};

} else { // 其它旧浏览器

utils.addListener = function (el, type, fn) {

el['on' + type] = fn;

};

utils.removeListener = function (el, type, fn) {

el['on' + type] = null;

};

}

用起来，是不是就很方便了？代码也优雅多了。

**自声明函数**

一般是在函数内部，重写同名函数代码，比如：

var scareMe = function () {

alert("Boo!");

scareMe = function () {

alert("Double boo!");

};

};

这种代码，非常容易使人迷惑，我们先来看看例子的执行结果：

// 1. 添加新属性

scareMe.property = "properly";

// 2. scareMe赋与一个新值

var prank = scareMe;

// 3. 作为一个方法调用

var spooky = {

boo: scareMe

};

// 使用新变量名称进行调用

prank(); // "Boo!"

prank(); // "Boo!"

console.log(prank.property); // "properly"

// 使用方法进行调用

spooky.boo(); // "Boo!"

spooky.boo(); // "Boo!"

console.log(spooky.boo.property); // "properly"

通过执行结果，可以发现，将定于的函数赋值与新变量（或内部方法），代码并不执行重载的scareMe代码，而如下例子则正好相反：

// 使用自声明函数

scareMe(); // Double boo!

scareMe(); // Double boo!

console.log(scareMe.property); // undefined

大家使用这种模式时，一定要非常小心才行，否则实际结果很可能和你期望的结果不一样，当然你也可以利用这个特殊做一些特殊的操作。

**内存优化**

该模式主要是利用函数的属性特性来避免大量的重复计算。通常代码形式如下：

var myFunc = function (param) {

if (!myFunc.cache[param]) {

var result = {};

// ... 复杂操作 ...

myFunc.cache[param] = result;

}

return myFunc.cache[param];

};

// cache 存储

myFunc.cache = {};

但是上述代码有个问题，如果传入的参数是toString或者其它类似Object拥有的一些公用方法的话，就会出现问题，这时候就需要使用传说中的hasOwnProperty方法了，代码如下：

var myFunc = function (param) {

if (!myFunc.cache.hasOwnProperty(param)) {

var result = {};

// ... 复杂操作 ...

myFunc.cache[param] = result;

}

return myFunc.cache[param];

};

// cache 存储

myFunc.cache = {};

或者如果你传入的参数是多个的话，可以将这些参数通过JSON的stringify方法生产一个cachekey值进行存储，代码如下：

var myFunc = function () {

var cachekey = JSON.stringify(Array.prototype.slice.call(arguments)),

result;

if (!myFunc.cache[cachekey]) {

result = {};

// ... 复杂操作 ...

myFunc.cache[cachekey] = result;

}

return myFunc.cache[cachekey];

};

// cache 存储

myFunc.cache = {};

或者多个参数的话，也可以利用arguments.callee特性：

var myFunc = function (param) {

var f = arguments.callee,

result;

if (!f.cache[param]) {

result = {};

// ... 复杂操作 ...

f.cache[param] = result;

}

return f.cache[param];

};

// cache 存储

myFunc.cache = {};