**[前端开发者进阶之惰性函数定义](http://www.cnblogs.com/pigtail/p/3442463.html)**

穆乙：<http://www.cnblogs.com/pigtail/p/3447660.html>

函数是js世界的一等公民，js的动态性、易变性在函数的应用上，体现的淋漓尽致。做为参数，做为返回值等，正是函数这些特性，使得js开发变的有趣。

下面就阐述一下，js一个有趣的应用--惰性函数定义（Lazy Function Definition）。

惰性载入表示函数执行的分支只会在函数第一次掉用的时候执行，在第一次调用过程中，该函数会被覆盖为另一个按照合适方式执行的函数，这样任何对原函数的调用就不用再经过执行的分支了。

下面我们看几个典型的例子：

[复制代码](javascript:void(0);)

function addEvent (type, element, fun) {

if (element.addEventListener) {

element.addEventListener(type, fun, false);

}

else if(element.attachEvent){

element.attachEvent('on' + type, fun);

}

else{

element['on' + type] = fun;

}

}

[复制代码](javascript:void(0);)

上面是注册函数监听的各浏览器兼容函数。由于，各浏览之间的差异，不得不在用的时候做能力检测。显然，单从功能上讲，已经做到了兼容浏览器。美中不足，每次绑定监听，都会对能力做一次检测。然而，真正的应用中，这显然是多余的，同一个应用环境中，其实只需要检测一次即可。

下面我们重写上面的addEvent：

[复制代码](javascript:void(0);)

function addEvent (type, element, fun) {

if (element.addEventListener) {

addEvent = function (type, element, fun) {

element.addEventListener(type, fun, false);

}

}

else if(element.attachEvent){

addEvent = function (type, element, fun) {

element.attachEvent('on' + type, fun);

}

}

else{

addEvent = function (type, element, fun) {

element['on' + type] = fun;

}

}

return addEvent(type, element, fun);

}

[复制代码](javascript:void(0);)

由上，第一次调用addEvent会对浏览器做能力检测，然后，重写了addEvent。下次再调用的时候，由于函数被重写，不会再做能力检测。

同样的应用，javascript高级程序设计里的一例子：

[复制代码](javascript:void(0);)

function createXHR(){

if (typeof XMLHttpRequest != "undefined"){

return new XMLHttpRequest();

} else if (typeof ActiveXObject != "undefined"){

if (typeof arguments.callee.activeXString != "string"){

var versions = ["MSXML2.XMLHttp.6.0", "MSXML2.XMLHttp.3.0",

"MSXML2.XMLHttp"];

for (var i=0,len=versions.length; i < len; i++){

try {

var xhr = new ActiveXObject(versions[i]);

arguments.callee.activeXString = versions[i];

return xhr;

} catch (ex){

//skip

}

}

}

return new ActiveXObject(arguments.callee.activeXString);

} else {

throw new Error("No XHR object available.");

}

}

[复制代码](javascript:void(0);)

很显然，惰性函数在这里优势更加明显，因为这里有更多的分支。下面我们看一下重写后台的函数：

[复制代码](javascript:void(0);)

function createXHR() {

if (typeof XMLHttpRequest != "undefined") {

createXHR = function () {

return new XMLHttpRequest();

}

return new XMLHttpRequest();

} else if (typeof ActiveXObject != "undefined") {

var curxhr;

var versions = ["MSXML2.XMLHttp.6.0", "MSXML2.XMLHttp.3.0",

"MSXML2.XMLHttp"];

for (var i = 0, len = versions.length; i < len; i++) {

try {

var xhr = new ActiveXObject(versions[i]);

curxhr = versions[i];

createXHR = function () {

return new ActiveXObject(curxhr);

}

return xhr;

} catch (ex) {

//skip

}

}

} else {

throw new Error("No XHR object available.");

}

}

[复制代码](javascript:void(0);)

浏览器之间最大的差异，莫过于Dom操作，Dom操作也是前端应用 中最频繁的操作，前端的大多性能提升，均体现在Dom操作方面。下面看一个Dom操作方面的惰性函数定义例子：

[复制代码](javascript:void(0);)

var getScrollY = function() {

if (typeof window.pageYOffset == 'number') {

getScrollY = function() {

return window.pageYOffset;

};

} else if ((typeof document.compatMode == 'string') &&

(document.compatMode.indexOf('CSS') >= 0) &&

(document.documentElement) &&

(typeof document.documentElement.scrollTop == 'number')) {

getScrollY = function() {

return document.documentElement.scrollTop;

};

} else if ((document.body) &&

(typeof document.body.scrollTop == 'number')) {

getScrollY = function() {

return document.body.scrollTop;

}

} else {

getScrollY = function() {

return NaN;

};

}

return getScrollY();

}

[复制代码](javascript:void(0);)

 惰性函数定义应用还体现在创建单例上：

[复制代码](javascript:void(0);)

unction Universe() {

// 缓存的实例

var instance = this;

// 其它内容

this.start\_time = 0;

this.bang = "Big";

// 重写构造函数

Universe = function () {

return instance;

};

}

[复制代码](javascript:void(0);)

当然，像上面这种例子有很多。惰性函数定义，应用场景我们可以总结一下：

1 应用频繁，如果只用一次，是体现不出它的优点出来的，用的次数越多，越能体现这种模式的优势所在；

2 固定不变，一次判定，在固定的应用环境中不会发生改变；

3 复杂的分支判断，没有差异性，不需要应用这种模式；