

深入理解JavaScript系列

看云文档小组



目 录

- (1) 编写高质量JavaScript代码的基本要点
- (2) 揭秘命名函数表达式
- (3) 全面解析Module模式
- (4) 立即调用的函数表达式
- (5) 强大的原型和原型链
- (6) S.O.L.I.D五大原则之单一职责SRP
- (7) S.O.L.I.D五大原则之开闭原则OCP
- (8) S.O.L.I.D五大原则之里氏替换原则LSP
- (9) 根本没有“JSON对象”这回事！
- (10) JavaScript核心（晋级高手必读篇）
- (11) 执行上下文（Execution Contexts）
- (12) 变量对象（Variable Object）
- (13) This? Yes, this!
- (14) 作用域链(Scope Chain)
- (15) 函数（Functions）
- (16) 闭包（Closures）
- (17) 面向对象编程之一般理论
- (18) 面向对象编程之ECMAScript实现
- (19) 求值策略
- (20) 《你真懂JavaScript吗？》答案详解
- (21) S.O.L.I.D五大原则之接口隔离原则ISP
- (22) S.O.L.I.D五大原则之依赖倒置原则DIP
- (23) JavaScript与DOM（上）——也适用于新手
- (24) JavaScript与DOM（下）
- (25) 设计模式之单例模式
- (26) 设计模式之构造函数模式
- (27) 设计模式之建造者模式
- (28) 设计模式之工厂模式
- (29) 设计模式之装饰者模式
- (30) 设计模式之外观模式
- (31) 设计模式之代理模式
- (32) 设计模式之观察者模式
- (33) 设计模式之策略模式
- (34) 设计模式之命令模式

- (35) 设计模式之迭代器模式
- (36) 设计模式之中介者模式
- (37) 设计模式之享元模式
- (38) 设计模式之职责链模式
- (39) 设计模式之适配器模式
- (40) 设计模式之组合模式
- (41) 设计模式之模板方法
- (42) 设计模式之原型模式
- (43) 设计模式之状态模式
- (44) 设计模式之桥接模式
- (45) 代码复用模式 (避免篇)
- (46) 代码复用模式 (推荐篇)
- (47) 对象创建模式 (上篇)
- (48) 对象创建模式 (下篇)
- (49) Function模式 (上篇)
- (50) Function模式 (下篇)
- (结局篇)

(1) 编写高质量JavaScript代码的基本要点

深入理解JavaScript系列文章，包括了原创，翻译，转载，整理等各类型文章，如果对你有帮助，请推荐支持一把，给大叔写作的动力。

来源：<http://www.cnblogs.com/TomXu/archive/2011/12/15/2288411.html>

作者：大叔

才华横溢的Stoyan Stefanov，在他写的由O’ Reilly初版的新书《JavaScript Patterns》(JavaScript模式)中，我想要是为我们的读者贡献其摘要，那会是件很美妙的事情。具体一点就是编写高质量JavaScript的一些要素，例如避免全局变量，使用单变量声明，在循环中预缓存length(长度)，遵循代码阅读，以及更多。

此摘要也包括一些与代码不太相关的习惯，但对整体代码的创建息息相关，包括撰写API文档、执行同行评审以及运行JSLint。这些习惯和最佳做法可以帮助你写出更好的，更易于理解和维护的代码，这些代码在几个月或是几年之后再回过头看看也是会觉得很自豪的。

书写可维护的代码(Writing Maintainable Code)

软件bug的修复是昂贵的，并且随着时间的推移，这些bug的成本也会增加，尤其当这些bug潜伏并慢慢出现在已经发布的软件中时。当你发现bug 的时候就立即修复它是最好的，此时你代码要解决的问题在你脑中还是很清晰的。否则，你转移到其他任务，忘了那个特定的代码，一段时间后再去查看这些代码就 需要：

- 花时间学习和理解这个问题
- 化时间是了解应该解决的问题代码

还有问题，特别对于大的项目或是公司，修复bug的这位伙计不是写代码的那个人（且发现bug和修复bug的不是同一个人）。因此，必须降低理解代码花费的时间，无论是一段时间前你自己写的代码还是团队中的其他成员写的代码。这关系到底线（营业收入）和开发人员的幸福，因为我们更应该去开发新的激动人心的事物而不是花几小时几天的时间去维护遗留代码。

另一个相关软件开发生命的事实是，读代码花费的时间要比写来得多。有时候，当你专注并深入思考某个问题的时候，你可以坐下来，一个下午写大量的代码。

你的代码很能很快就工作了，但是，随着应用的成熟，还会有很多其他的事情发生，这就要

求你的进行进行审查，修改，和调整。例如：

- bug是暴露的
- 新功能被添加到应用程序
- 程序在新的环境下工作（例如，市场上出现新想浏览器）
- 代码改变用途
- 代码得完全从头重新，或移植到另一个架构上或者甚至使用另一种语言

由于这些变化，很少人力数小时写的代码最终演变成花数周来阅读这些代码。这就是为什么创建可维护的代码对应用程序的成功至关重要。

可维护的代码意味着：

- 可读的
- 一致的
- 可预测的
- 看上去就像是同一个人写的
- 已记录

最小全局变量(Minimizing Globals)

JavaScript通过函数管理作用域。在函数内部声明的变量只在这个函数内部，函数外面不可用。另一方面，全局变量就是在任何函数外面声明的或是未声明直接简单使用的。

每个JavaScript环境有一个全局对象，当你在任意的函数外面使用this的时候可以访问到。你创建的每一个全部变量都成了这个全局对象的属性。在浏览器中，方便起见，该全局对象有个附加属性叫做window，此window(通常)指向该全局对象本身。下面的代码片段显示了如何在浏览器环境中创建和访问的全局变量：

```
myglobal = "hello"; // 不推荐写法
console.log(myglobal); // "hello"
console.log(window.myglobal); // "hello"
console.log(window["myglobal"]); // "hello"
console.log(this.myglobal); // "hello"
```

全局变量的问题

全局变量的问题在于，你的JavaScript应用程序和web页面上的所有代码都共享了这些全局变量，他们住在同一个全局命名空间，所以当程序的两个不同部分定义同名但不同作用的全局变量的时候，命名冲突在所难免。

web页面包含不是该页面开发者所写的代码也是比较常见的，例如：

- 第三方的JavaScript库
- 广告方的脚本代码
- 第三方用户跟踪和分析脚本代码
- 不同类型的小组件，标志和按钮

比方说，该第三方脚本定义了一个全局变量，叫做result；接着，在你的函数中也定义一个名为result的全局变量。其结果就是后面的变量覆盖前面的，第三方脚本就一下子囫屁啦！

因此，要想和其他脚本成为好邻居的话，尽可能少的使用全局变量是很重要的。在书中后面提到的一些减少全局变量的策略，例如命名空间模式或是函数立即自动执行，但是要想让全局变量少最重要的还是始终使用var来声明变量。

由于JavaScript的两个特征，不自觉地创建出全局变量是出乎意料的容易。首先，你可以甚至不需要声明就可以使用变量；第二，JavaScript有隐含的全局概念，意味着你不声明的任何变量都会成为一个全局对象属性。参考下面的代码：

```
function sum(x, y) {  
    // 不推荐写法：隐式全局变量  
    result = x + y;  
    return result;  
}
```

此段代码中的 result 没有声明。代码照样运作正常，但在调用函数后你最后的结果就多一个全局命名空间，这可以是一个问题的根源。

经验法则是始终使用var声明变量，正如改进版的sum()函数所演示的：

```
function sum(x, y) {  
    var result = x + y;  
    return result;  
}
```

另一个创建隐式全局变量的反例就是使用任务链进行部分var声明。下面的片段中，a 是本地变量但是 b 确实全局变量，这可能不是你希望发生的：

```
// 反例，勿使用  
function foo() {  
    var a = b = 0;  
    // ...  
}
```

```
}
```

此现象发生的原因在于这个从右到左的赋值，首先，是赋值表达式 `b = 0`，此情况下b是未声明的。这个表达式的返回值是0，然后这个0就分配给了通过var定义的这个局部变量a。换句话说，就好比输入了：

```
var a = (b = 0);
```

如果你已经准备好声明变量，使用链分配是比较好的做法，不会产生任何意料之外的全局变量，如：

```
function foo() {  
    var a, b;  
    // ... a = b = 0; // 两个均局部变量  
}
```

然而，另外一个避免全局变量的原因是可移植性。如果你想你的代码在不同的环境下（主机下）运行，使用全局变量如履薄冰，因为你会无意中覆盖你最初环境下不存在的主机对象（所以你原以为名称可以放心大胆地使用，实际上对于有些情况并不适用）。

忘记var的副作用(Side Effects When Forgetting var)

隐式全局变量和明确定义的全局变量间有些小的差异，就是通过 `delete` 操作符让变量未定义的能力。

- 通过var创建的全局变量（任何函数之外的程序中创建）是不能被删除的。
- 无var创建的隐式全局变量（无视是否在函数中创建）是能被删除的。

这表明，在技术上，隐式全局变量并不是真正的全局变量，但它们是全局对象的属性。属性是可以通过 `delete` 操作符删除的，而变量是不能的：

```
// 定义三个全局变量  
var global_var = 1;  
global_novar = 2; // 反面教材  
(function () {  
    global_fromfunc = 3; // 反面教材  
})();  
  
// 试图删除  
delete global_var; // false  
delete global_novar; // true
```

```
delete global_fromfunc; // true

// 测试该删除
typeof global_var; // "number"
typeof global_novar; // "undefined"
typeof global_fromfunc; // "undefined"
```

在ES5严格模式下，未声明的变量（如在前面的代码片段中的两个反面教材）工作时会抛出一个错误。

访问全局对象(Access to the Global Object)

在浏览器中，全局对象可以通过 `window` 属性在代码的任何位置访问（除非你做了些比较出格的事情，像是声明了一个名为`window`的局部变量）。但是在其他环境下，这个方便的属性可能被叫做其他什么东西（甚至在程序中不可用）。如果你需要在没有硬编码的 `window` 标识符下访问全局对象，你可以在任何层级的函数作用域中做如下操作：

```
var global = (function () {
    return this;
})();
```

这种方法可以随时获得全局对象，因为其在函数中被当做函数调用了（不是通过 `new` 构造），`this` 总是指向全局对象。实际上这个病不适用于ECMAScript 5严格模式，所以，在严格模式下时，你必须采取不同的形式。例如，你正在开发一个JavaScript库，你可以将你的代码包裹在一个即时函数中，然后从全局作用域中，传递一个引用指向`this`作为你即时函数的参数。

单var形式 (Single var Pattern)

在函数顶部使用单var语句是比较有用的一种形式，其好处在于：

- 提供了一个单一的地方去寻找功能所需要的所有局部变量
- 防止变量在定义之前使用的逻辑错误
- 帮助你记住声明的全局变量，因此较少了全局变量//zxx:此处我自己是有点晕乎的...
- 少代码（类型啊传值啊单线完成）

单var形式长得就像下面这个样子：

```
function func() {
    var a = 1,
        b = 2,
        sum = a + b,
        myobject = {},
```



```

        i,
        j;
    // function body...
}

```

您可以使用一个var语句声明多个变量，并以逗号分隔。像这种初始化变量同时初始化值的做法是很好的。这样子可以防止逻辑错误（所有未初始化但声明的变量的初始值是 `undefined`）和增加代码的可读性。在你看到代码后，你可以根据初始化的值知道这些变量大致的用途，例如是要当作对象呢还是当作整数来使。

你也可以在声明的时候做一些实际的工作，例如前面代码中的 `sum = a + b` 这个情况，另外一个例子就是当你使用DOM（文档对象模型）引用时，你可以使用单一的var把DOM引用一起指定为局部变量，就如下面代码所示的：

```

function updateElement() {
    var el = document.getElementById("result"),
        style = el.style;
    // 使用el和style干点其他什么事...
}

```

预解析：var散布的问题(Hoisting: A Problem with Scattered vars)

JavaScript中，你可以在函数的任何位置声明多个var语句，并且它们就好像是在函数顶部声明一样发挥作用，这种行为称为 `hoisting`（悬置/置顶解析/预解析）。当你使用了一个变量，然后不久在函数中又重新声明的话，就可能产生逻辑错误。对于JavaScript，只要你的变量是在同一个作用域中（同一函数），它都被当做是声明的，即使是它在var声明前使用的时候。看下面这个例子：

```

// 反例
myname = "global"; // 全局变量
function func() {
    alert(myname); // "undefined"
    var myname = "local";
    alert(myname); // "local"
}
func();

```

在这个例子中，你可能会以为第一个alert弹出的是“global”，第二个弹出“local”。这种期许是可以理解的，因为在第一个alert的时候，myname未声明，此时函数肯定很自然而然地看全局变量myname，但是，实际上并不是这么工作的。第一个alert会弹

出“ undefined” 是因为myname被当做了函数的局部变量（ 尽管是之后声明的 ），所有的变量声明当被悬置到函数的顶部了。因此，为了避免这种混乱，最好是预先声明你想使用的全部变量。

上面的代码片段执行的行为可能就像下面这样：

```
myname = "global"; // global variable
function func() {
    var myname; // 等同于 -> var myname = undefined;
    alert(myname); // "undefined"
    myname = "local";
    alert(myname); // "local"}
func();
```

为了完整，我们再提一提执行层面的稍微复杂点的东西。代码处理分两个阶段，第一阶段是变量，函数声明，以及正常格式的参数创建，这是一个解析和进入上下文的阶段。第二个阶段是代码执行，函数表达式和不合格的标识符（为声明的变量）被创建。但是，出于实用的目的，我们就采用了“ hoisting” 这个概念，这种ECMAScript标准中并未定义，通常用来描述行为。

for循环(for Loops)

在 for 循环中，你可以循环取得数组或是数组类似对象的值，譬如 arguments 和 HTMLCollection 对象。通常的循环形式如下：

```
// 次佳的循环
for (var i = 0; i < myarray.length; i++) {
    // 使用myarray[i]做点什么
}
```

这种形式的循环的不足在于每次循环的时候数组的长度都要去获取下。这回降低你的代码，尤其当 myarray 不是数组，而是一个 HTMLCollection 对象的时候。

HTMLCollections 指的是DOM方法返回的对象，例如：

```
document.getElementsByName()
document.getElementsByClassName()
document.getElementsByTagName()
```

还有其他一些 HTMLCollections ，这些是在DOM标准之前引进并且现在还在使用的。

有：

```
document.images：页面上所有的图片元素
document.links  ：所有a标签元素
document.forms  ：所有表单
document.forms[0].elements ：页面上第一个表单中的所有域
```

集合的麻烦在于它们实时查询基本文档（HTML页面）。这意味着每次你访问任何集合的长度，你要实时查询DOM，而DOM操作一般都是比较昂贵的。

这就是为什么当你循环获取值时，缓存数组(或集合)的长度是比较好的形式，正如下面代码显示的：

```
for (var i = 0, max = myarray.length; i < max; i++) {
    // 使用myarray[i]做点什么
}
```

这样，在这个循环过程中，你只检索了一次长度值。

在所有浏览器下，循环获取内容时缓存 HTMLCollections 的长度是更快的，2倍(Safari3)到190倍(IE7)之间。//zxx:此数据貌似很老，仅供参考

注意到，当你明确想要修改循环中的集合的时候（例如，添加更多的DOM元素），你可能更喜欢长度更新而不是常量。

伴随着单var形式，你可以把变量从循环中提出来，就像下面这样：

```
function looper() {
    var i = 0,
        max,
        myarray = [];
    // ...
    for (i = 0, max = myarray.length; i < max; i++) {
        // 使用myarray[i]做点什么
    }
}
```

这种形式具有一致性的好处，因为你坚持了单一var形式。不足在于当重构代码的时候，复制和粘贴整个循环有点困难。例如，你从一个函数复制了一个循环到另一个函数，你不得不去确定你能够把 i 和 max 引入新的函数（如果在这里没有用的话，很有可能你要从原函数中把它们删掉）。

最后一个需要对循环进行调整的是使用下面表达式之一来替换 `i++` 。

```
i = i + 1  
i += 1
```

JSLint提示您这样做，原因是 `++` 和 `--` 促进了“过分棘手(excessive trickiness)” 。//zxx:
这里比较难翻译，我想本意应该是让代码变得更加的棘手
如果你直接无视它，JSLint的 `plusplus` 选项会是 `false` （默认是default）。

还有两种变化的形式，其又有了些微改进，因为：

- 少了一个变量(无max)
- 向下数到0，通常更快，因为和0做比较要比和数组长度或是其他不是0的东西作比较更有效率

//第一种变化的形式：

```
var i, myarray = [];  
for (i = myarray.length; i--;) {  
    // 使用myarray[i]做点什么  
}
```

//第二种使用while循环：

```
var myarray = [],  
    i = myarray.length;  
while (i--) {  
    // 使用myarray[i]做点什么  
}
```

这些小的改进只体现在性能上，此外JSLint会对使用*i--*加以抱怨。

for-in循环(for-in Loops)

`for-in` 循环应该用在非数组对象的遍历上，使用 `for-in` 进行循环也被称为“枚举”。

从技术上讲，你可以使用 `for-in` 循环数组（因为JavaScript中数组也是对象），但这是不推荐的。因为如果数组对象已被自定义的功能增强，就可能发生逻辑错误。另外，在for-in中，属性列表的顺序（序列）是不能保证的。所以最好数组使用正常的for循环，对象使用for-in循环。

有个很重要的 `hasOwnProperty()` 方法，当遍历对象属性的时候可以过滤掉从原型链上下来的属性。

思考下面一段代码：

```
// 对象
var man = {
    hands: 2,
    legs: 2,
    heads: 1
};

// 在代码的某个地方
// 一个方法添加给了所有对象
if (typeof Object.prototype.clone === "undefined") {
    Object.prototype.clone = function () {};
}
```

在这个例子中，我们有一个使用对象字面量定义的名叫man的对象。在man定义完成后的某个地方，在对象原型上增加了一个很有用的名叫 clone()的方法。此原型链是实时的，这就意味着所有的对象自动可以访问新的方法。为了避免枚举man的时候出现clone()方法，你需要应用 hasOwnProperty() 方法过滤原型属性。如果不做过滤，会导致clone()函数显示出来，在大多数情况下这是不希望出现的。

```
// 1.
// for-in 循环
for (var i in man) {
    if (man.hasOwnProperty(i)) { // 过滤
        console.log(i, ":", man[i]);
    }
}
/* 控制台显示结果
hands : 2
legs : 2
heads : 1*/
// 2.
// 反面例子:
// for-in loop without checking hasOwnProperty()
for (var i in man) {
    console.log(i, ":", man[i]);
}
/*
控制台显示结果
hands : 2
legs : 2
heads : 1
clone: function()
*/
```

另外一种使用 hasOwnProperty() 的形式是取消Object.prototype上的方法。像是：

```
for (var i in man) {  
    if (Object.prototype.hasOwnProperty.call(man, i)) { // 过滤  
        console.log(i, ":", man[i]);  
    }  
}
```

其好处在于在man对象重新定义hasOwnProperty情况下避免命名冲突。也避免了长属性查找对象的所有方法，你可以使用局部变量“缓存”它。

```
var i, hasOwn = Object.prototype.hasOwnProperty;  
for (i in man) {  
    if (hasOwn.call(man, i)) { // 过滤  
        console.log(i, ":", man[i]);  
    }  
}
```

严格来说，不使用 hasOwnProperty() 并不是一个错误。根据任务以及你对代码的自信程度，你可以跳过它以提高些许的循环速度。但是当你当前对象内容（和其原型链）不确定的时候，添加 hasOwnProperty() 更加保险些。

格式化的变化（通不过JSLint）会直接忽略掉花括号，把if语句放到同一行上。其优点在于循环语句读起来就像一个完整的想法（每个元素都有一个自己的属性“X”，使用“X”干什么）：

```
// 警告：通不过JSLint检测  
var i, hasOwn = Object.prototype.hasOwnProperty;  
for (i in man) if (hasOwn.call(man, i)) { // 过滤  
    console.log(i, ":", man[i]);  
}
```

（不）扩展内置原型((Not) Augmenting Built-in Prototypes)

扩增构造函数的prototype属性是个很强大的增加功能的方法，但有时候它太强大了。

增加内置的构造函数原型（如Object(), Array(), 或Function()）挺诱人的，但是这严重降低了可维护性，因为它让你的代码变得难以预测。使用你代码的其他开发人员很可能更期望使用内置的JavaScript方法来持续不断地工作，而不是你另加的方法。

另外，属性添加到原型中，可能会导致不使用hasOwnProperty属性时在循环中显示出来，这会造成混乱。

因此，不增加内置原型是最好的。你可以指定一个规则，仅当下面的条件均满足时例外：

- 可以预期将来的ECMAScript版本或是JavaScript实现将一直将此功能当作内置方法来实现。例如，你可以添加ECMAScript 5中描述的方法，一直到各个浏览器都迎头赶上。这种情况下，你只是提前定义了有用的方法。
- 如果您检查您的自定义属性或方法已不存在——也许已经在代码的其他地方实现或已经是你支持的浏览器JavaScript引擎部分。
- 你清楚地文档记录并和团队交流了变化。

如果这三个条件得到满足，你可以给原型进行自定义的添加，形式如下：

```
if (typeof Object.prototype.myMethod !== "function") {  
    Object.prototype.myMethod = function () {  
        // 实现...  
    };  
}
```

switch模式(switch Pattern)

你可以通过类似下面形式的switch语句增强可读性和健壮性：

```
var inspect_me = 0,  
    result = '';  
switch (inspect_me) {  
case 0:  
    result = "zero";  
    break;  
case 1:  
    result = "one";  
    break;  
default:  
    result = "unknown";  
}
```

这个简单的例子中所遵循的风格约定如下：

- 每个case和switch对齐（花括号缩进规则除外）
- 每个case中代码缩进
- 每个case以break清除结束
- 避免贯穿（故意忽略break）。如果你非常确信贯穿是最好的方法，务必记录此情况，因为对于有些阅读人而言，它们可能看起来是错误的。
- 以default结束switch：确保总有健全的结果，即使无情况匹配。

避免隐式类型转换(Avoiding Implied Typecasting)

JavaScript的变量在比较的时候会隐式类型转换。这就是为什么一些诸如：`false == 0` 或 `"" == 0` 返回的结果是`true`。为避免引起混乱的隐含类型转换，在你比较值和表达式类型的时候始终使用`===`和`!==`操作符。

```
var zero = 0;
if (zero === false) {
    // 不执行，因为zero为0，而不是false
}

// 反面示例
if (zero == false) {
    // 执行了...
}
```

还有另外一种思想观点认为`==`就足够了`===`是多余的。例如，当你使用`typeof`你就知道它会返回一个字符串，所以没有使用严格相等的理由。然而，JSLint要求严格相等，它使代码看上去更有一致性，可以降低代码阅读时的精力消耗。（“`==`是故意的还是一个疏漏？”）

避免(Avoiding) eval()

如果你现在的代码中使用了`eval()`，记住该咒语“`eval()`是魔鬼”。此方法接受任意的字符串，并当作JavaScript代码来处理。当有问题的代码是事先知道的（不是运行时确定的），没有理由使用`eval()`。如果代码是在运行时动态生成，有一个更好的方式不使用`eval`而达到同样的目标。例如，用方括号表示法来访问动态属性会更好更简单：

```
// 反面示例
var property = "name";
alert(eval("obj." + property));

// 更好的
var property = "name";
alert(obj[property]);
```

使用`eval()`也带来了安全隐患，因为被执行的代码（例如从网络来）可能已被篡改。这是个很常见的反面教材，当处理Ajax请求得到的JSON相应的时候。在这些情况下，最好使用JavaScript内置方法来解析JSON相应，以确保安全和有效。若浏览器不支持`JSON.parse()`，你可以使用来自JSON.org的库。

同样重要的是要记住，给`setInterval()`、`setTimeout()`和`Function()`构造函数传递字符串，大部分情况下，与使用`eval()`是类似的，因此要避免。在幕后，JavaScript仍需要评估和执行你

给程序传递的字符串：

```
// 反面示例
setTimeout("myFunc()", 1000);
setTimeout("myFunc(1, 2, 3)", 1000);

// 更好的
setTimeout(myFunc, 1000);
setTimeout(function () {
    myFunc(1, 2, 3);
}, 1000);
```

使用新的Function()构造就类似于eval()，应小心接近。这可能是一个强大的构造，但往往被误用。如果你绝对必须使用eval()，你可以考虑使用new Function()代替。有一个小的潜在好处，因为在新Function()中作代码评估是在局部函数作用域中运行，所以代码中任何被评估的通过var 定义的变量都不会自动变成全局变量。另一种方法来阻止自动全局变量是封装eval()调用到一个即时函数中。

考虑下面这个例子，这里仅 un 作为全局变量污染了命名空间。

```
console.log(typeof un);    // "undefined"
console.log(typeof deux); // "undefined"
console.log(typeof trois); // "undefined"

var jsstring = "var un = 1; console.log(un);";
eval(jsstring); // logs "1"

jsstring = "var deux = 2; console.log(deux);";
new Function(jsstring)(); // logs "2"

jsstring = "var trois = 3; console.log(trois);";
(function () {
    eval(jsstring);
})(); // logs "3"

console.log(typeof un); // number
console.log(typeof deux); // "undefined"
console.log(typeof trois); // "undefined"
```

另一间eval()和Function构造不同的是eval()可以干扰作用域链，而Function()更安分守己些。不管你在哪里执行Function()，它只看到全局作用域。所以其能很好的避免本地变量污染。在下面这个例子中，eval()可以访问和修改它外部作用域中的变量，这是Function做不来的（注意到使用Function和new Function是相同的）。

```
(function () {
```

本文档使用 [看云](#) 构建

```

    var local = 1;
    eval("local = 3; console.log(local)"); // logs "3"
    console.log(local); // logs "3"
  }());

  (function () {
    var local = 1;
    Function("console.log(typeof local);")(); // logs undefined
  }());

```

parseInt()下的数值转换(Number Conversions with parseInt())

使用parseInt()你可以从字符串中获取数值，该方法接受另一个基数参数，这经常省略，但不应该。当字符串以“0”开头的时候就有可能会出问题，例如，部分时间进入表单域，在ECMAScript 3中，开头为“0”的字符串被当做8进制处理了，但这已在ECMAScript 5中改变了。为了避免矛盾和意外的结果，总是指定基数参数。

```

var month = "06",
    year = "09";
month = parseInt(month, 10);
year = parseInt(year, 10);

```

此例中，如果你忽略了基数参数，如parseInt(year)，返回的值将是0，因为“09”被当做8进制（好比执行parseInt(year, 8)），而09在8进制中不是个有效数字。

替换方法是将字符串转换成数字，包括：

```

+"08" // 结果是 8
Number("08") // 8

```

这些通常快于parseInt()，因为parseInt()方法，顾名思义，不是简单地解析与转换。但是，如果你想输入例如“08 hello”，parseInt()将返回数字，而其它以NaN告终。

编码规范(Coding Conventions)

建立和遵循编码规范是很重要的，这让你的代码保持一致性，可预测，更易于阅读和理解。一个新的开发者加入这个团队可以通读规范，理解其它团队成员书写的代码，更快上手干活。

许多激烈的争论发生会议上或是邮件列表上，问题往往针对某些代码规范的特定方面（例如代码缩进，是Tab制表符键还是space空格键）。如果你是 你组织中建议采用规范的，准备

好面对各种反对的或是听起来不同但很强烈的观点。要记住，建立和坚定不移地遵循规范要比纠结于规范的细节重要的多。

缩进(Indentation)

代码没有缩进基本上就不能读了。唯一糟糕的事情就是不一致的缩进，因为它看上去像是遵循了规范，但是可能一路上伴随着混乱和惊奇。重要的是规范地使用缩进。

一些开发人员更喜欢用tab制表符缩进，因为任何人都可以调整他们的编辑器以自己喜欢的空格数来显示Tab。有些人喜欢空格——通常四个，这都无所谓，只要团队每个人都遵循同一个规范就好了。这本书，例如，使用四个空格缩进，这也是JSLint中默认的缩进。

什么应该缩进呢？规则很简单——花括号里面的东西。这就意味着函数体，循环 (do, while, for, for-in), if, switch, 以及对象字面量中的对象属性。下面的代码就是使用缩进的示例：

```
function outer(a, b) {
    var c = 1,
        d = 2,
        inner;
    if (a > b) {
        inner = function () {
            return {
                r: c - d
            };
        };
    } else {
        inner = function () {
            return {
                r: c + d
            };
        };
    }
    return inner;
}
```

花括号{}(Curly Braces)

花括号（亦称大括号，下同）应总被使用，即使在它们为可选的时候。技术上讲，在in或是for中如果语句仅一条，花括号是不需要的，但是你还是应该总是使用它们，这会让代码更有持续性和易于更新。

想象下你有一个只有一条语句的for循环，你可以忽略花括号，而没有解析的错误。

```
// 糟糕的实例
for (var i = 0; i < 10; i += 1)
    alert(i);
```

但是，如果，后来，主体循环部分又增加了行代码？

```
// 糟糕的实例
for (var i = 0; i < 10; i += 1)
    alert(i);
    alert(i + " is " + (i % 2 ? "odd" : "even"));
```

第二个alert已经在循环之外，缩进可能欺骗了你。为了长远打算，最好总是使用花括号，即时值一行代码：

```
// 好的实例
for (var i = 0; i < 10; i += 1) {
    alert(i);
}
```

if条件类似：

```
// 坏
if (true)
    alert(1);
else
    alert(2);

// 好
if (true) {
    alert(1);
} else {
    alert(2);
}
```

左花括号的位置(Opening Brace Location)

开发人员对于左大括号的位置有着不同的偏好——在同一行或是下一行。

```
if (true) {
    alert("It's TRUE!");
}

//或

if (true)
{
    alert("It's TRUE!");
}
```

这个实例中，仁者见仁智者见智，但也有个案，括号位置不同会有不同的行为表现。这是因为分号插入机制(semicolon insertion mechanism)——JavaScript是不挑剔的，当你选择不使用分号结束一行代码时JavaScript会自己帮你补上。这种行为可能会导致麻烦，如当你返回对象字面量，而左括号却在下一行的时候：

```
// 警告： 意外的返回值
function func() {
  return
  // 下面代码不执行
  {
    name : "Batman"
  }
}
```

如果你希望函数返回一个含有name属性的对象，你会惊讶。由于隐含分号，函数返回undefined。前面的代码等价于：

```
// 警告： 意外的返回值
function func() {
  return undefined;
  // 下面代码不执行
  {
    name : "Batman"
  }
}
```

总之，总是使用花括号，并始终把在与之之前的语句放在同一行：

```
function func() {
  return {
    name : "Batman"
  };
}
```

关于分号注：就像使用花括号，你应该总是使用分号，即使他们可由JavaScript解析器隐式创建。这不仅促进更科学和更严格的代码，而且有助于解决存有疑惑的地方，就如前面的例子显示。

空格(White Space)

空格的使用同样有助于改善代码的可读性和一致性。在写英文句子的时候，在逗号和句号后面会使用间隔。在JavaScript中，你可以按照同样的逻辑在列表模样表达式（相当于逗号）

和结束语句（相对于完成了“想法”）后面添加间隔。

适合使用空格的地方包括：

- for循环分号分开后的部分：如 `for (var i = 0; i < 10; i += 1) {...}`
- for循环中初始化的多变量(i和max)：
`for (var i = 0, max = 10; i < max; i += 1) {...}`
- 分隔数组项的逗号的后面：`var a = [1, 2, 3];`
- 对象属性逗号的后面以及分隔属性名和属性值的冒号的后面：
`var o = {a: 1, b: 2};`
- 限定函数参数：`myFunc(a, b, c)`
- 函数声明的花括号的前面：`function myFunc() {}`
- 匿名函数表达式function的后面：`var myFunc = function () {};`

使用空格分开所有的操作符和操作对象是另一个不错的使用，这意味着在

`+`, `-`, `*`, `=`, `,`, `==`, `!=="`, `&&`, `||`, `+=` 等前后都需要空格。

```
// 宽松一致的间距
// 使代码更易读
// 使得更加“透气”
var d = 0,
    a = b + 1;
if (a && b && c) {
    d = a % c;
    a += d;
}

// 反面例子
// 缺失或间距不一
// 使代码变得疑惑
var d = 0,
    a = b + 1;
if (a&&b&&c) {
    d=a % c;
    a+= d;
}
```

最后需要注意的一个空格——花括号间距。最好使用空格：

- 函数、if-else语句、循环、对象字面量的左花括号的前面(`{`)
- else或while之间的右花括号(`}`)

空格使用的一点不足就是增加了文件的大小，但是压缩无此问题。

有一个经常被忽略的代码可读性方面是垂直空格的使用。你可以使用空行来分隔代码单元，就像是文学作品中使用段落分隔一样。

命名规范(Naming Conventions)

另一种方法让你的代码更具可预测性和可维护性是采用命名规范。这就意味着你需要用同一种形式给你的变量和函数命名。

下面是建议的一些命名规范，你可以原样采用，也可以根据自己的喜好作调整。同样，遵循规范要比规范是什么更重要。

以大写字母写构造函数(Capitalizing Constructors)

JavaScript并没有类，但有new调用的构造函数：

```
var adam = new Person();
```

因为构造函数仍仅仅是函数，仅看函数名就可以帮助告诉你这应该是一个构造函数还是一个正常的函数。

命名构造函数时首字母大写具有暗示作用，使用小写命名的函数和方法不应该使用new调用：

```
function MyConstructor() {...}  
function myFunction() {...}
```

分隔单词(Separating Words)

当你的变量或是函数名有多个单词的时候，最好单词的分离遵循统一的规范，有一个常见的做法被称作“驼峰(Camel)命名法”，就是单词小写，每个单词的首字母大写。

对于构造函数，可以使用大驼峰式命名法(upper camel case)，如 `MyConstructor()`。对于函数和方法名称，你可以使用小驼峰式命名法(lower camel case)，像是 `myFunction()`，`calculateArea()` 和 `getFirstName()`。

要是变量不是函数呢？开发者通常使用小驼峰式命名法，但还有另外一种做法就是所有单词小写以下划线连接：例如，`first_name`，`favorite_bands`，和 `old_company_name`，这种标记法帮你直观地区分函数和其他标识——原型和对象。

ECMAScript的属性和方法均使用Camel标记法，尽管多字的属性名称是罕见的（正则表达

式对象的lastIndex和ignoreCase属性)。

其它命名形式(Other Naming Patterns)

有时，开发人员使用命名规范来弥补或替代语言特性。

例如，JavaScript中没有定义常量的方法（尽管有些内置的像Number, MAX_VALUE），所以开发者都采用全部单词大写的规范来命名这个程序生命周期中都不会改变的变量，如：

```
// 珍贵常数，只可远观
var PI = 3.14,
    MAX_WIDTH = 800;
```

还有另外一个完全大写的惯例：全局变量名字全部大写。全部大写命名全局变量可以加强减小全局变量数量的实践，同时让它们易于区分。

另外一种使用规范来模拟功能的是私有成员。虽然可以在JavaScript中实现真正的私有，但是开发者发现仅仅使用一个下划线前缀来表示一个私有属性或方法会更容易些。考虑下面的例子：

```
var person = {
  getName: function () {
    return this._getFirst() + ' ' + this._getLast();
  },
  _getFirst: function () {
    // ...
  },
  _getLast: function () {
    // ...
  }
};
```

在此例中，getName() 就表示公共方法，部分稳定的API。而 _getFirst() 和 _getLast() 则表明了私有。它们仍然是正常的公共方法，但是使用下划线前缀来警告person对象的使用者这些方法在下一个版本中时不能保证工作的，是不能直接使用的。注意，JSLint有些不鸟下划线前缀，除非你设置了noman选项为:false。

下面是一些常见的_private规范：

- 使用尾下划线表示私有，如name和getElements()
- 使用一个下划线前缀表_protected（保护）属性，两个下划线前缀表示__private（私有）属性

- Firefox中一些内置的变量属性不属于该语言的技术部分，使用两个前下划线和两个后下划线表示，如：`__proto__`和`__parent__`。

注释(Writing Comments)

你必须注释你的代码，即使不会其他人向你一样接触它。通常，当你深入研究一个问题，你会很清楚的知道这个代码是干嘛用的，但是，当你一周之后再回来查看的时候，想必也要耗掉不少脑细胞去搞明白到底怎么工作的。

很显然，注释不能走极端：每个单独变量或是单独一行。但是，你通常应该记录所有的函数，它们的参数和返回值，或是任何不寻常的技术和方法。要想到注释可以给你代码未来的读者以诸多提示；读者需要的是（不要读太多的东西）仅注释和函数属性名来理解你的代码。例如，当你有五六行程序执行特定的任务，如果你提供了一行代码目的以及为什么在这里的描述的话，读者就可以直接跳过这段细节。没有硬性规定注释代码比，代码的某些部分（如正则表达式）可能注释 要比代码多。

最重要的习惯，然而也是最难遵守的，就是保持注释的及时更新，因为过时的注释比没有注释更加的误导人。

关于作者 (About the Author)

Stoyan Stefanov是Yahoo!web开发人员，多个O'Reilly书籍的作者、投稿者和技术评审。他经常在会议和他的博客www.phpied.com上发表web开发主题的演讲。Stoyan还是smush.it图片优化工具的创造者，YUI贡献者，雅虎性能优化工具YSlow 2.0的架构设计师。

本文转自：<http://www.zhangxinxu.com/wordpress/?p=1173>

英文原文：<http://net.tutsplus.com/tutorials/javascript-ajax/the-essentials-of-writing-high-quality-javascript/>

(2) 揭秘命名函数表达式

前言

网上还没用发现有人对命名函数表达式进去重复深入的讨论，正因为如此，网上出现了各种各样的误解，本文将从原理和实践两个方面来探讨JavaScript关于命名函数表达式的优缺点。

简单的说，命名函数表达式只有一个用户，那就是在Debug或者Profiler分析的时候来描述函数的名称，也可以使用函数名实现递归，但很快你就会发现其实是不切实际的。当然，如果你不关注调试，那就没什么可担心的了，否则，如果你想了解兼容性方面的东西的话，你还是应该继续往下看。

我们先开始看看，什么叫函数表达式，然后再说一下现代调试器如何处理这些表达式，如果你已经对这方面很熟悉的话，请直接跳过此小节。

函数表达式和函数声明

在ECMAScript中，创建函数的最常用的两个方法是函数表达式和函数声明，两者期间的区别是有点晕，因为ECMA规范只明确了一点：函数声明必须带有标示符（Identifier）（就是大家常说的函数名称），而函数表达式则可以省略这个标示符：

函数声明:

```
function 函数名称 (参数：可选){ 函数体 }
```

函数表达式：

```
function 函数名称 ( 可选 ) (参数：可选){ 函数体 }
```

所以，可以看出，如果不声明函数名称，它肯定是表达式，可如果声明了函数名称的话，如何判断是函数声明还是函数表达式呢？ECMAScript是通过上下文来区分的，如果function foo(){}是作为赋值表达式的一部分的话，那它就是一个函数表达式，如果function foo(){}被包含在一个函数体内，或者位于程序的最顶部的话，那它就是一个函数声明。

```
function foo(){} // 声明，因为它是程序的一部分
var bar = function foo(){}; // 表达式，因为它是赋值表达式的一部分

new function bar(){}; // 表达式，因为它是new表达式

(function(){
function bar(){} // 声明，因为它是函数体的一部分
})();
```

还有一种函数表达式不太常见，就是被括号括住的(function foo()){})，他是表达式的原因是因为括号 ()是一个分组操作符，它的内部只能包含表达式，我们来看几个例子：

```
function foo(){} // 函数声明
(function foo()){}; // 函数表达式：包含在分组操作符内

try {
  (var x = 5); // 分组操作符，只能包含表达式而不能包含语句：这里的var就是语句
} catch(err) {
  // SyntaxError
}
```

你可以会想到，在使用eval对JSON进行执行的时候，JSON字符串通常被包含在一个圆括号里：eval('(' + json + ')')，这样做的原因就是因为分组操作符，也就是这对括号，会让解析器强制将JSON的花括号解析成表达式而不是代码块。

```
try {
  { "x": 5 }; // "{" 和 "}" 做解析成代码块
} catch(err) {
  // SyntaxError
}

({ "x": 5 }); // 分组操作符强制将 "{" 和 "}"作为对象字面量来解析
```

表达式和声明存在着十分微妙的差别，首先，函数声明会在任何表达式被解析和求值之前先被解析和求值，即使你的声明在代码的最后一行，它也会在同作用域内第一个表达式之前被解析/求值，参考如下例子，函数fn是在alert之后声明的，但是在alert执行的时候，fn已经有定义了：

```
alert(fn());

function fn() {
  return 'Hello world!';
}
```

另外，还有一点需要提醒一下，函数声明在条件语句内虽然可以用，但是没有被标准化，也就是说不同的环境可能有不同的执行结果，所以这样情况下，最好使用函数表达式：

```
// 千万别这样做！
// 因为有的浏览器会返回first的这个function，而有的浏览器返回的却是第二个

if (true) {
  function foo() {
    return 'first';
  }
}
else {
  function foo() {
    return 'second';
  }
}
foo();

// 相反，这样情况，我们要用函数表达式
var foo;
if (true) {
  foo = function() {
    return 'first';
  };
}
else {
  foo = function() {
    return 'second';
  };
}
foo();
```

函数声明的实际规则如下：

函数声明只能出现在程序或函数体内。从句法上讲，它们不能出现在Block（块）（{ ... }）中，例如不能出现在 if、while 或 for 语句中。因为 Block（块）中只能包含 Statement 语句，而不能包含 函数声明 这样的源元素。另一方面，仔细看一看规则也会发现，唯一可能让 表达式 出现在 Block（块）中情形，就是让它作为 表达式语句 的一部分。但是，规范明确规定了 表达式语句 不能以关键字 function 开头。而这实际上就是说，函数表达式 同样也不能出现在 Statement 语句或 Block（块）中（因为 Block（块）就是由 Statement 语句构成的）。

函数语句

在 ECMAScript 的语法扩展中，有一个是函数语句，目前只有基于 Gecko 的浏览器实现了该扩展，所以对于下面的例子，我们仅是抱着学习的目的来看，一般来说不推荐使用（除非你针

对Gecko浏览器进行开发)。

1.一般语句能用的地方，函数语句也能用，当然也包括Block块中：

```
if (true) {  
  function f(){ }  
}  
else {  
  function f(){ }  
}
```

2.函数语句可以像其他语句一样被解析，包含基于条件执行的情形

```
if (true) {  
  function foo(){ return 1; }  
}  
else {  
  function foo(){ return 2; }  
}  
foo(); // 1  
// 注：其它客户端会将foo解析成函数声明  
// 因此，第二个foo会覆盖第一个，结果返回2，而不是1
```

3.函数语句不是在变量初始化期间声明的，而是在运行时声明的——与函数表达式一样。不过，函数语句的标识符一旦声明能在函数的整个作用域生效了。标识符有效性正是导致函数语句与函数表达式不同的关键所在（下一小节我们将会展示命名函数表达式的具体行为）。

```
// 此刻，foo还没用声明  
typeof foo; // "undefined"  
if (true) {  
  // 进入这里以后，foo就被声明在整个作用域内了  
  function foo(){ return 1; }  
}  
else {  
  // 从来不会走到这里，所以这里的foo也不会被声明  
  function foo(){ return 2; }  
}  
typeof foo; // "function"
```

不过，我们可以使用下面这样的符合标准的代码来模式上面例子中的函数语句：

```
var foo;  
if (true) {  
  foo = function foo(){ return 1; };  
}
```

```
else {  
  foo = function foo() { return 2; };  
}
```

4.函数语句和函数声明（或命名函数表达式）的字符串表示类似，也包括标识符：

```
if (true) {  
  function foo(){ return 1; }  
}  
String(foo); // function foo() { return 1; }
```

5.另外一个，早期基于Gecko的实现（Firefox 3及以前版本）中存在一个bug，即函数语句覆盖函数声明的方式不正确。在这些早期的实现中，函数语句不知何故不能覆盖函数声明：

```
// 函数声明  
function foo(){ return 1; }  
if (true) {  
  // 用函数语句重写  
  function foo(){ return 2; }  
}  
foo(); // FF3以下返回1, FF3.5以上返回2  
  
// 不过，如果前面是函数表达式，则没用问题  
var foo = function(){ return 1; };  
if (true) {  
  function foo(){ return 2; }  
}  
foo(); // 所有版本都返回2
```

再次强调一点，上面这些例子只是在某些浏览器支持，所以推荐大家不要使用这些，除非你就在特性的浏览器上做开发。

命名函数表达式

函数表达式在实际应用中还是很常见的，在web开发中有个常用的模式是基于对某种特性的测试来伪装函数定义，从而达到性能优化的目的，但由于这种方式都是在同一作用域内，所以基本上一定要用函数表达式：

```
// 该代码来自Garrett Smith的APE Javascript library库(http://dhtmlkitchen.com/ape/)  
var contains = (function() {  
  var docEl = document.documentElement;  
  
  if (typeof docEl.compareDocumentPosition !== 'undefined') {  
    return function(e1, b) {  
      return (e1.compareDocumentPosition(b) & 16) !== 0;  
    };  
  }  
})
```

```

    };
  }
  else if (typeof docEl.contains !== 'undefined') {
    return function(el, b) {
      return el !== b && el.contains(b);
    };
  }
  return function(el, b) {
    if (el === b) return false;
    while (el !== b && (b = b.parentNode) !== null);
    return el === b;
  };
})();

```

提到命名函数表达式，理所当然，就是它得有名字，前面的例子`var bar = function foo(){};`就是一个有效的命名函数表达式，但有一点需要记住：这个名字只在新定义的函数作用域内有效，因为规范规定了标示符不能在外围的作用域内有效：

```

var f = function foo(){
  return typeof foo; // foo是在内部作用域内有效
};
// foo在外部用于是不可见的
typeof foo; // "undefined"
f(); // "function"

```

既然，这么要求，那命名函数表达式到底有啥用啊？为啥要取名？

正如我们开头所说：给它一个名字就是可以让调试过程更方便，因为在调试的时候，如果在调用栈中的每个项都有自己的名字来描述，那么调试过程就太爽了，感受不一样嘛。

调试器中的函数名

如果一个函数有名字，那调试器在调试的时候会将它的名字显示在调用的栈上。有些调试器（Firebug）有时候还会为你们函数取名并显示，让他们和那些应用该函数的便利具有相同的角色，可是通常情况下，这些调试器只安装简单的规则来取名，所以说没有太大价格，我们来看一个例子：

```

function foo(){
  return bar();
}
function bar(){
  return baz();
}
function baz(){
  debugger;
}
foo();

```

```
// 这里我们使用了3个带名字的函数声明
// 所以当调试器走到debugger语句的时候，Firebug的调用栈上看起来非常清晰明了
// 因为很明白地显示了名称
baz
bar
foo
expr_test.html()
```

通过查看调用栈的信息，我们可以很明了地知道foo调用了bar, bar又调用了baz（而foo本身有在expr_test.html文档的全局作用域内被调用），不过，还有一个比较爽地方，就是刚才说的Firebug为匿名表达式取名的功能：

```
function foo(){
return bar();
}
var bar = function(){
return baz();
}
function baz(){
debugger;
}
foo();

// Call stack
baz
bar() //看到了么？
foo
expr_test.html()
```

然后，当函数表达式稍微复杂一些的时候，调试器就不那么聪明了，我们只能在调用栈中看到问号：

```
function foo(){
return bar();
}
var bar = (function(){
if (window.addEventListener) {
return function(){
return baz();
};
}
else if (window.attachEvent) {
return function() {
return baz();
};
}
})();
function baz(){
```



```
debugger;
}
foo();

// Call stack
baz
(?)() // 这里可是问号哦
foo
expr_test.html()
```

另外，当把函数赋值给多个变量的时候，也会出现令人郁闷的问题：

```
function foo(){
return baz();
}
var bar = function(){
debugger;
};
var baz = bar;
bar = function() {
alert('spoofed');
};
foo();

// Call stack:
bar()
foo
expr_test.html()
```

这时候，调用栈显示的是foo调用了bar，但实际上并非如此，之所以有这种问题，是因为baz和另外一个包含alert('spoofed')的函数做了引用交换所导致的。

归根结底，只有给函数表达式取个名字，才是最委托的办法，也就是使用命名函数表达式。我们来使用带名字的表达式来重写上面的例子（注意立即调用的表达式块里返回的2个函数的名字都是bar）：

```
function foo(){
return bar();
}
var bar = (function(){
if (window.addEventListener) {
return function bar(){
return baz();
};
}
else if (window.attachEvent) {
return function bar() {
return baz();
};
};
```

```
}  
})();  
function baz(){  
  debugger;  
}  
foo();  
  
// 又再次看到了清晰的调用栈信息了耶!  
baz  
bar  
foo  
expr_test.html()
```

OK，又学了一招吧？不过在高兴之前，我们再看看不同寻常的JScript吧。

JScript的Bug

比较恶的是，IE的ECMAScript实现JScript严重混淆了命名函数表达式，搞得现很多人都出来反对命名函数表达式，而且即便是最新的一版（IE8中使用的5.8版）仍然存在下列问题。

下面我们就来看看IE在实现中究竟犯了那些错误，俗话说知己知彼，才能百战不殆。我们来看看如下几个例子：

例1：函数表达式的标示符泄露到外部作用域

```
var f = function g(){};  
typeof g; // "function"
```

上面我们说过，命名函数表达式的标示符在外部作用域是无效的，但JScript明显是违反了这一规范，上面例子中的标示符g被解析成函数对象，这就乱了套了，很多难以发现的bug都是因为这个原因导致的。

注：IE9貌似已经修复了这个问题

例2：将命名函数表达式同时当作函数声明和函数表达式

```
typeof g; // "function"  
var f = function g(){};
```

特性环境下，函数声明会优先于任何表达式被解析，上面的例子展示的是JScript实际上是把命名函数表达式当成函数声明了，因为它在实际声明之前就解析了g。

这个例子引出了下一个例子。

例3：命名函数表达式会创建两个截然不同的函数对象！

```
var f = function g(){};
f === g; // false

f.expando = 'foo';
g.expando; // undefined
```

看到这里，大家会觉得问题严重了，因为修改任何一个对象，另外一个没有什么改变，这太恶了。通过这个例子可以发现，创建2个不同的对象，也就是说如果你想修改f的属性中保存某个信息，然后想当然地通过引用相同对象的g的同名属性来使用，那问题就大了，因为根本就不可能。

再来看一个稍微复杂的例子：

例4：仅仅顺序解析函数声明而忽略条件语句块

```
var f = function g() {
  return 1;
};
if (false) {
  f = function g(){
    return 2;
  };
}
g(); // 2
```

这个bug查找就难多了，但导致bug的原因却非常简单。首先，g被当作函数声明解析，由于JScript中的函数声明不受条件代码块约束，所以在这个很恶的if分支中，g被当作另一个函数function g(){ return 2 }，也就是又被声明了一次。然后，所有“常规的”表达式被求值，而此时f被赋予了另一个新创建的对象引用。由于在对表达式求值的时候，永远不会进入“这个可恶if分支，因此f就会继续引用第一个函数function g(){ return 1 }。分析到这里，问题就很清楚了：假如你不够细心，在f中调用了g，那么将会调用一个毫不相干的g函数对象。

你可能会问，将不同的对象和arguments.callee相比较时，有什么样的区别呢？我们来看看：

```
var f = function g(){
  return [
    arguments.callee == f,
    arguments.callee == g
```

```

];
};
f(); // [true, false]
g(); // [false, true]

```

可以看到，arguments.callee的引用一直是被调用的函数，实际上这也是好事，稍后会解释。

还有一个有趣的例子，那就是在不包含声明的赋值语句中使用命名函数表达式：

```

(function(){
  f = function f(){};
})();

```

按照代码的分析，我们原本是想创建一个全局属性f（注意不要和一般的匿名函数混淆了，里面用的是带名字的生命），JScript在这里捣乱了一把，首先他把表达式当成函数声明解析了，所以左边的f被声明为局部变量了（和一般的匿名函数里的声明一样），然后在函数执行的时候，f已经是定义过的了，右边的function f(){}则直接就赋值给局部变量f了，所以f就不是全局属性。

了解了JScript这么变态以后，我们就要及时预防这些问题了，首先防范标识符泄漏带外部作用域，其次，应该永远不引用被用作函数名称的标识符；还记得前面例子中那个讨人厌的标识符g吗？——如果我们能够当g不存在，可以避免多少不必要的麻烦哪。因此，关键就在于始终要通过f或者arguments.callee来引用函数。如果你使用了命名函数表达式，那么应该只在调试的时候利用那个名字。最后，还要记住一点，一定要把命名函数表达式声明期间错误创建的函数清理干净。

对于，上面最后一点，我们还得再解释一下。

JScript的内存管理

知道了这些不符合规范的代码解析bug以后，我们如果用它的话，就会发现内存方面其实是有问题的，来看一个例子：

```

var f = (function(){
  if (true) {
    return function g(){};
  }
  return function g(){};
})();

```

我们知道，这个匿名函数调用返回的函数（带有标识符g的函数），然后赋值给了外部的f。我们也知道，命名函数表达式会导致产生多余的函数对象，而该对象与返回的函数对象不是一回事。所以这个多余的g函数就死在了返回函数的闭包中了，因此内存问题就出现了。这是因为if语句内部的函数与g是在同一个作用域中被声明的。这种情况下，除非我们显式断开对g函数的引用，否则它一直占着内存不放。

```
var f = (function(){
  var f, g;
  if (true) {
    f = function g(){};
  }
  else {
    f = function g(){};
  }
  // 设置g为null以后它就不会再占内存了
  g = null;
  return f;
})();
```

通过设置g为null，垃圾回收器就把g引用的那个隐式函数给回收掉了，为了验证我们的代码，我们来做一些测试，以确保我们的内存被回收了。

测试

测试很简单，就是命名函数表达式创建10000个函数，然后把它们保存在一个数组中。等一会儿以后再看这些函数到底占用了多少内存。然后，再断开这些引用并重复这一过程。下面是测试代码：

```
function createFn(){
  return (function(){
    var f;
    if (true) {
      f = function F(){
        return 'standard';
      };
    }
    else if (false) {
      f = function F(){
        return 'alternative';
      };
    }
    else {
      f = function F(){
        return 'fallback';
      };
    }
    // var F = null;
  })();
}
```

```

    return f;
  })();
}

var arr = [ ];
for (var i=0; i<10000; i++) {
  arr[i] = createFn();
}

```

通过运行在Windows XP SP2中的任务管理器可以看到如下结果：

IE6:

```

without `null`: 7.6K -> 20.3K
with `null`:    7.6K -> 18K

```

IE7:

```

without `null`: 14K -> 29.7K
with `null`:    14K -> 27K

```

如我们所料，显示断开引用可以释放内存，但是释放的内存不是很多，10000个函数对象才释放大约3M的内存，这对一些小型脚本不算什么，但对于大型程序，或者长时间运行在低内存的设备里的时候，这是非常有必要的。

关于在Safari 2.x中JS的解析也有一些bug，但介于版本比较低，所以我们在这里就不介绍了，大家如果想看的话，请仔细查看英文资料。

SpiderMonkey的怪癖

大家都知道，命名函数表达式的标识符只在函数的局部作用域中有效。但包含这个标识符的局部作用域又是什么样子的吗？其实非常简单。在命名函数表达式被求值时，会创建一个特殊的对象，该对象的唯一目的就是保存一个属性，而这个属性的名字对应着函数标识符，属性的值对应着那个函数。这个对象会被注入到当前作用域链的前端。然后，被“扩展”的作用域链又被用于初始化函数。

在这里，有一点十分有意思，那就是ECMA-262定义这个（保存函数标识符的）“特殊”对象的方式。标准说“像调用new Object()表达式那样”创建这个对象。如果从字面上来理解这句话，那么这个对象就应该是全局Object的一个实例。然而，只有一个实现是按照标准字面上的要求这么做的，这个实现就是SpiderMonkey。因此，在SpiderMonkey中，扩展Object.prototype有可能会干扰函数的局部作用域：

```
Object.prototype.x = 'outer';
```

```

(function(){
    var x = 'inner';

    /*函数foo的作用域链中有一个特殊的对象——用于保存函数的标识符。这个特殊的对象实际上就是{
    foo:  }。
    当通过作用域链解析x时，首先解析的是foo的局部环境。如果没有找到x，则继续搜索作用域链
    中的下一个对象。下一个对象
    就是保存函数标识符的那个对象——{ foo:  }，由于该对象继承自Object.prototype，所以
    在此可以找到x。
    而这个x的值也就是Object.prototype.x的值（outer）。结果，外部函数的作用域（包含x
    = 'inner'的作用域）就不会被解析了。 */

    (function foo()){
        alert(x); // 提示框中显示：outer
    })();
})();

```

不过，更高版本的SpiderMonkey改变了上述行为，原因可能是认为那是一个安全漏洞。也就是说，“特殊”对象不再继承Object.prototype了。不过，如果你使用Firefox 3或者更低版本，还可以“重温”这种行为。

另一个把内部对象实现为全局Object对象的是黑莓（Blackberry）浏览器。目前，它的活动对象（Activation Object）仍然继承Object.prototype。可是，ECMA-262并没有说_活动对象_也要“像调用new Object()表达式那样”来创建（或者说像创建保存NFE标识符的对象一样创建）。人家规范只说了_活动对象_是规范中的一种机制。

那我们就来看看黑莓里都发生了什么：

```

Object.prototype.x = 'outer';

(function(){
    var x = 'inner';

    (function(){
        /*在沿着作用域链解析x的过程中，首先会搜索局部函数的活动对象。当然，在该对象中找不到x
        。
        可是，由于活动对象继承自Object.prototype，因此搜索x的下一个目标就是Object.prototype；而
        Object.prototype中又确实有x的定义。结果，x的值就被解析为——outer。跟前面的例子差不多，
        包含x = 'inner'的外部函数的作用域（活动对象）就不会被解析了。 */

        alert(x); // 显示：outer
    })();
})();

```

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

不过神奇的还是，函数中的变量甚至会与已有的 `Object.prototype` 的成员发生冲突，来看看下面的代码：

```
(function(){  
  var constructor = function(){ return 1; };  
  
  (function(){  
    constructor(); // 求值结果是{}（即相当于调用了Object.prototype.constructor()  
    ) 而不是1  
  
    constructor === Object.prototype.constructor; // true  
    toString === Object.prototype.toString; // true  
  
    // .....  
  
  })();  
})();
```

要避免这个问题，要避免使用`Object.prototype`里的属性名称，如`toString`, `valueOf`, `hasOwnProperty`等等。

JScrip解决方案

```
var fn = (function(){  
  
  // 声明要引用函数的变量  
  var f;  
  
  // 有条件地创建命名函数  
  // 并将其引用赋值给f  
  if (true) {  
    f = function F(){ }  
  }  
  else if (false) {  
    f = function F(){ }  
  }  
  else {  
    f = function F(){ }  
  }  
  
  // 声明一个与函数名（标识符）对应的变量，并赋值为null  
  // 这实际上是给相应标识符引用的函数对象作了一个标记，  
  // 以便垃圾回收器知道可以回收它了
```



```
var F = null;

// 返回根据条件定义的函数
return f;
})();
```

最后我们给出一个应用上述技术的应用实例，这是一个跨浏览器的addEvent函数代码：

```
// 1) 使用独立的作用域包含声明
var addEvent = (function(){

var docEl = document.documentElement;

// 2) 声明要引用函数的变量
var fn;

if (docEl.addEventListener) {

    // 3) 有意给函数一个描述性的标识符
    fn = function addEvent(element, eventName, callback) {
        element.addEventListener(eventName, callback, false);
    }
} else if (docEl.attachEvent) {
    fn = function addEvent(element, eventName, callback) {
        element.attachEvent('on' + eventName, callback);
    }
} else {
    fn = function addEvent(element, eventName, callback) {
        element['on' + eventName] = callback;
    }
}

// 4) 清除由JScript创建的addEvent函数
// 一定要保证在赋值前使用var关键字
// 除非函数顶部已经声明了addEvent
var addEvent = null;

// 5) 最后返回由fn引用的函数
return fn;
})();
```

替代方案

其实，如果我们不想要这个描述性名字的话，我们就可以用最简单的形式来做，也就是在函数内部声明一个函数（而不是函数表达式），然后返回该函数：

```
var hasClassName = (function(){

    // 定义私有变量

本文档使用 看云 构建
```

```

var cache = { };

// 使用函数声明
function hasClassName(element, className) {
    var _className = '(?:^|\\s+)' + className + '(?:\\s+|$)';
    var re = cache[_className] || (cache[_className] = new RegExp(_className));
    return re.test(element.className);
}

// 返回函数
return hasClassName;
})();

```

显然，当存在多个分支函数定义时，这个方案就不行了。不过有种模式貌似可以实现：那就是提前使用函数声明来定义所有函数，并分别为这些函数指定不同的标识符：

```

var addEvent = (function(){
    var docEl = document.documentElement;

    function addEventListener(){
        /* ... */
    }
    function attachEvent(){
        /* ... */
    }
    function addEventAsProperty(){
        /* ... */
    }

    if (typeof docEl.addEventListener != 'undefined') {
        return addEventListener;
    }
    elseif (typeof docEl.attachEvent != 'undefined') {
        return attachEvent;
    }
    return addEventAsProperty;
})();

```

虽然这个方案很优雅，但也不是没有缺点。第一，由于使用不同的标识符，导致丧失了命名的一致性。且不说这样好还是坏，最起码它不够清晰。有人喜欢使用相同的名字，但也有人根本不在乎字眼上的差别。可毕竟，不同的名字会让人联想到所用的不同实现。例如，在调试器中看到attachEvent，我们就知道 addEvent 是基于 attachEvent 的实现。当然，基于实现来命名的方式也不一定都行得通。假如我们要提供一个API，并按照这种方式把函数命名为inner。那么API用户的很容易就会被相应实现的 细节搞得晕头转向。

要解决这个问题，当然就得想一套更合理的命名方案了。但关键是不要再额外制造麻烦。我
本文档使用 [看云](#) 构建

现在能想起来的方案大概有如下几个：

```
'addEvent', 'altAddEvent', 'fallbackAddEvent'
// 或者
'addEvent', 'addEvent2', 'addEvent3'
// 或者
'addEvent_addEventListener', 'addEvent_attachEvent', 'addEvent_asProperty',
```

另外，这种模式还存在一个小问题，即增加内存占用。提前创建N个不同名字的函数，等于有N-1的函数是用不到的。具体来讲，如果 `document.documentElement` 中包含 `attachEvent`，那么 `addEventListener` 和 `addEventAsProperty` 则根本就用不着了。可是，他们都占着内存哪；而且，这些内存将永远都得不到释放，原因跟JScript臭哄哄的命名表达式相同——这两个函数都被“截留”在返回的那个函数的闭包中了。

不过，增加内存占用这个问题确实没什么大不了的。如果某个库——例如Prototype.js——采用了这种模式，无非也就是多创建一两百个函数而已。只要不是（在运行时）重复地创建这些函数，而是只（在加载时）创建一次，那么就没有什么好担心的。

WebKit的displayName

WebKit团队在这个问题采取了有点儿另类的策略。介于匿名和命名函数如此之差的表现力，WebKit引入了一个“特殊的” `displayName` 属性（本质上是一个字符串），如果开发人员为函数的这个属性赋值，则该属性的值将在调试器或性能分析器中被显示在函数“名称”的位置上。[Francisco Tolmasky详细地解释了这个策略的原理和实现](#)。

未来考虑

将来的ECMAScript-262第5版（目前还是草案）会引入所谓的严格模式（strict mode）。开启严格模式的实现会禁用语言中的那些不稳定、不可靠和不安全的特性。据说出于安全方面的考虑，`arguments.callee` 属性将在严格模式下被“封杀”。因此，在处于严格模式时，访问 `arguments.callee` 会导致 `TypeError`（参见ECMA-262第5版的10.6节）。而我之所以在此提到严格模式，是因为如果在基于第5版标准的实现中无法使用 `arguments.callee` 来执行递归操作，那么使用命名函数表达式的可能性就会大大增加。从这个意义上来说，理解命名函数表达式的语义及其bug也就显得更加重要了。

```
// 此前，你可能会使用arguments.callee
(function(x) {
  if (x return 1;
  return x * arguments.callee(x - 1);
})(10);
```

```
// 但在严格模式下，有可能就要使用命名函数表达式
(function factorial(x) {
  if (x return 1;
  return x * factorial(x - 1);
})(10);

// 要么就退一步，使用没有那么灵活的函数声明
function factorial(x) {
  if (x return 1;
  return x * factorial(x - 1);
}
factorial(10);
```

致谢

理查德·康福德 (Richard Cornford)，是他率先[解释了JScript中命名函数表达式所存在的bug](#)。理查德解释了我在这篇文章中提及的大多数bug，所以我强烈建议大家去看看他的解释。我还要感谢Yann-Erwan Perio和道格拉斯·克劳克佛德 (Douglas Crockford)，他们早在2003年就在[comp.lang.javascript论坛中提及并讨论NFE问题了](#)。

约翰·戴维·道尔顿 (John-David Dalton) 对 “最终解决方案” 提出了很好的建议。

托比·兰吉的点子被我用在 “替代方案” 中。

盖瑞特·史密斯 (Garrett Smith) 和德米特里·苏斯尼科 (Dmitry Soshnikov) 对本文的多方面作出了补充和修正。

英文原文：<http://kangax.github.com/nfe/>

参考译文：[连接访问](#) (SpiderMonkey的怪癖之后的章节参考该文)

(3) 全面解析Module模式

简介

Module模式是JavaScript编程中一个非常通用的模式，一般情况下，大家都知道基本用法，本文尝试着给大家更多该模式的高级使用方式。

首先我们来看看Module模式的基本特征：

1. 模块化，可重用
2. 封装了变量和function，和全局的namespace不接触，松耦合
3. 只暴露可用public的方法，其它私有方法全部隐藏

关于Module模式，最早是由YUI的成员Eric Miraglia在4年前提出了这个概念，我们将从一个简单的例子来解释一下基本的用法（如果你已经非常熟悉了，请忽略这一节）。

基本用法

先看一下最简单的一个实现，代码如下：

```
var Calculator = function (eq) {  
    //这里可以声明私有成员  
  
    var eqCtl = document.getElementById(eq);  
  
    return {  
        // 暴露公开的成员  
        add: function (x, y) {  
            var val = x + y;  
            eqCtl.innerHTML = val;  
        }  
    };  
};
```

我们可以通过如下的方式来调用：

```
var calculator = new Calculator('eq');  
calculator.add(2, 2);
```

大家可能看到了，每次用的时候都要new一下，也就是说每个实例在内存里都是一份copy，如果你不需要传参数或者没有一些特殊苛刻的要求的话，我们可以在最后一个}后面加上一个

括号，来达到自执行的目的，这样该实例在内存中只会存在一份copy，不过在展示他的优点之前，我们还是先来看看这个模式的基本使用方法吧。

匿名闭包

匿名闭包是让一切成为可能的基础，而这也是JavaScript最好的特性，我们来创建一个最简单的闭包函数，函数内部的代码一直存在于闭包内，在整个运行周期内，该闭包都保证了内部的代码处于私有状态。

```
(function () {
    // ... 所有的变量和function都在这里声明，并且作用域也只能在这个匿名闭包里
    // ...但是这里的代码依然可以访问外部全局的对象
})();
```

注意，匿名函数后面的括号，这是JavaScript语言所要求的，因为如果你不声明的话，JavaScript解释器默认是声明一个function函数，有括号，就是创建一个函数表达式，也就是自执行，用的时候不用和上面那样在new了，当然你也可以这样来声明：

```
(function () { /* 内部代码 */ })();
```

不过我们推荐使用第一种方式，关于函数自执行，我后面会有专门一篇文章进行详解，这里就不多说了。

引用全局变量

JavaScript有一个特性叫做隐式全局变量，不管一个变量有没有用过，JavaScript解释器反向遍历作用域链来查找整个变量的var声明，如果没有找到var，解释器则假定该变量是全局变量，如果该变量用于了赋值操作的话，之前如果不存在的话，解释器则会自动创建它，这就是说在匿名闭包里使用或创建全局变量非常容易，不过比较困难的是，代码比较难管理，尤其是阅读代码的人看着很多区分哪些变量是全局的，哪些是局部的。

不过，好在在匿名函数里我们可以提供一个比较简单的替代方案，我们可以将全局变量当成一个参数传入到匿名函数然后使用，相比隐式全局变量，它又清晰又快，我们来看一个例子：

```
(function ($, YAHOO) {
    // 这里，我们的代码就可以使用全局的jQuery对象了，YAHOO也是一样
})(jQuery, YAHOO));
```

现在很多类库里都有这种使用方式，比如jQuery源码。

本文档使用 [看云](#) 构建

不过，有时候可能不仅仅要使用全局变量，而是也想声明全局变量，如何做呢？我们可以通过匿名函数的返回值来返回这个全局变量，这也就是一个基本的Module模式，来看一个完整的代码：

```
var blogModule = (function () {
    var my = {}, privateName = "博客园";

    function privateAddTopic(data) {
        // 这里是内部处理代码
    }

    my.Name = privateName;
    my.AddTopic = function (data) {
        privateAddTopic(data);
    };

    return my;
} ());
```

上面的代码声明了一个全局变量blogModule，并且带有2个可访问的属性：blogModule.AddTopic和blogModule.Name，除此之外，其它代码都在匿名函数的闭包里保持着私有状态。同时根据上面传入全局变量的例子，我们也可以很方便地传入其它的全局变量。

高级用法

上面的内容对大多数用户已经很足够了，但我们还可以基于此模式延伸出更强大，易于扩展的结构，让我们一个一个来看。

扩展

Module模式的一个限制就是所有的代码都要写在一个文件，但是在一些大型项目里，将一个功能分离成多个文件是非常重要的，因为可以多人合作易于开发。再回头看看上面的全局参数导入例子，我们能否把blogModule自身传进去呢？答案是肯定的，我们先将blogModule传进去，添加一个函数属性，然后再返回就达到了我们所说的目的，上代码：

```
var blogModule = (function (my) {
    my.AddPhoto = function () {
        //添加内部代码
    };
    return my;
} (blogModule));
```

这段代码，看起来是不是有C#里扩展方法的感觉？有点类似，但本质不一样哦。同时尽管

var不是必须的，但为了确保一致，我们再次使用了它，代码执行以后，blogModule下的AddPhoto就可以使用了，同时匿名函数内部的代码也依然保证了私密性和内部状态。

松耦合扩展

上面的代码尽管可以执行，但是必须先声明blogModule，然后再执行上面的扩展代码，也就是说步骤不能乱，怎么解决这个问题呢？我们来回想一下，我们平时声明变量的都是都是这样的：

```
var cnblogs = cnblogs || {} ;
```

这是确保cnblogs对象，在存在的时候直接用，不存在的时候直接赋值为{}，我们来看看如何利用这个特性来实现Module模式的任意加载顺序：

```
var blogModule = (function (my) {
    // 添加一些功能

    return my;
} (blogModule || {}));
```

通过这样的代码，每个单独分离的文件都保证这个结构，那么我们就可以实现任意顺序的加载，所以，这个时候的var就是必须要声明的，因为不声明，其它文件读取不到哦。

紧耦合扩展

虽然松耦合扩展很牛叉了，但是可能也会存在一些限制，比如你没办法重写你的一些属性或者函数，也不能在初始化的时候就是用Module的属性。紧耦合扩展限制了加载顺序，但是提供了我们重载的机会，看如下例子：

```
var blogModule = (function (my) {
    var oldAddPhotoMethod = my.AddPhoto;

    my.AddPhoto = function () {
        // 重载方法，依然可通过oldAddPhotoMethod调用旧的方法
    };

    return my;
} (blogModule));
```

通过这种方式，我们达到了重载的目的，当然如果你想在继续在内部使用原有的属性，你可以调用oldAddPhotoMethod来用。

克隆与继承

```
var blogModule = (function (old) {
    var my = {},
        key;

    for (key in old) {
        if (old.hasOwnProperty(key)) {
            my[key] = old[key];
        }
    }

    var oldAddPhotoMethod = old.AddPhoto;
    my.AddPhoto = function () {
        // 克隆以后，进行了重写，当然也可以继续调用oldAddPhotoMethod
    };

    return my;
} (blogModule));
```

这种方式灵活是灵活，但是也需要花费灵活的代价，其实该对象的属性对象或function根本没有被复制，只是对同一个对象多了一种引用而已，所以如果老对象去改变它，那克隆以后的对象所拥有的属性或function函数也会被改变，解决这个问题，我们就得是用递归，但递归对function函数的赋值也不好，所以我们在递归的时候eval相应的function。不管怎样，我还是把这方式放在这个帖子里了，大家使用的时候注意一下就行了。

跨文件共享私有对象

通过上面的例子，我们知道，如果一个module分割到多个文件的话，每个文件需要保证一样的结构，也就是说每个文件匿名函数里的私有对象都不能交叉访问，那如果我们非要使用，那怎么办呢？我们先看一段代码：

```
var blogModule = (function (my) {
    var _private = my._private = my._private || {},

        _seal = my._seal = my._seal || function () {
            delete my._private;
            delete my._seal;
            delete my._unseal;
        },

        _unseal = my._unseal = my._unseal || function () {
            my._private = _private;
            my._seal = _seal;
            my._unseal = _unseal;
        };

    return my;
} (blogModule));
```

```
} (blogModule || {}));
```

任何文件都可以对他们的局部变量`_private`设属性，并且设置对其他的文件也立即生效。一旦这个模块加载结束，应用会调用 `blogModule._seal()` "上锁"，这会阻止外部接入内部的`_private`。如果这个模块需要再次增生，应用的生命周期内，任何文件都可以调用`_unseal()` "开锁"，然后再加载新文件。加载后再次调用`_seal()` "上锁"。

子模块

最后一个也是最简单的使用方式，那就是创建子模块

```
blogModule.CommentSubModule = (function () {
    var my = {};
    // ...

    return my;
} ());
```

尽管非常简单，我还是把它放进来了，因为我想说明的是子模块也具有一般模块所有的高级使用方式，也就是说你可以对任意子模块再次使用上面的一些应用方法。

总结

上面的大部分方式都可以互相组合使用的，一般来说如果要设计系统，可能会用到松耦合扩展，私有状态和子模块这样的方式。另外，我这里没有提到性能问题，但我认为Module模式效率高，代码少，加载速度快。使用松耦合扩展允许并行加载，这更可以提升下载速度。不过初始化时间可能要慢一些，但是为了使用好的模式，这是值得的。

参考文章：

<http://yuiblog.com/blog/2007/06/12/module-pattern/>

<http://www.adequatelygood.com/2010/3/JavaScript-Module-Pattern-In-Depth>

(4) 立即调用的函数表达式

前言

大家学JavaScript的时候，经常遇到自执行匿名函数的代码，今天我们主要就来想想说一下自执行。

在详细了解这个之前，我们来谈了解一下“自执行”这个叫法，本文对这个功能的叫法也不一定完全对，主要是看个人如何理解，因为有的人说立即调用，有的人说自动执行，所以你可以完全按照你自己的理解来取一个名字，不过我听很多人都叫它为“自执行”，但作者后面说了很多，来说服大家称呼为“立即调用的函数表达式”。

本文英文原文地址：<http://benalman.com/news/2010/11/immediately-invoked-function-expression/>

什么是自执行？

在JavaScript里，任何function在执行的时候都会创建一个执行上下文，因为为function声明的变量和function有可能只在该function内部，这个上下文，在调用function的时候，提供了一种简单的方式来创建自由变量或私有子function。

```
// 由于该function里返回了另外一个function，其中这个function可以访问自由变量i
// 所有说，这个内部的function实际上是有权限可以调用内部的对象。
```

```
function makeCounter() {
    // 只能在makeCounter内部访问i
    var i = 0;

    return function () {
        console.log(++i);
    };
}
```

```
// 注意，counter和counter2是不同的实例，分别有自己范围内的i。
```

```
var counter = makeCounter();
counter(); // logs: 1
counter(); // logs: 2
```

```
var counter2 = makeCounter();
counter2(); // logs: 1
counter2(); // logs: 2
```

```
alert(i); // 引用错误：i没有defined（因为i是存在于makeCounter内部）。
```

很多情况下，我们不需要makeCounter多个实例，甚至某些case下，我们也不需要显示的返回值，OK，往下看。

问题的核心

当你声明类似function foo(){}或var foo = function(){}函数的时候，通过在后面加个括弧就可以实现自执行，例如foo()，看代码：

```
// 因为想下面第一个声明的function可以在后面加一个括弧()就可以自己执行了，比如foo(),
// 因为foo仅仅是function() { /* code */ }这个表达式的一个引用

var foo = function(){ /* code */ }

// ...是不是意味着后面加个括弧都可以自动执行？

function(){ /* code */ }(); // SyntaxError: Unexpected token (
//
```

上述代码，如果甚至运行，第2个代码会出错，因为在解析器解析全局的function或者function内部function关键字的时候，默认是认为function声明，而不是function表达式，如果你不显示告诉编译器，它默认会声明成一个缺少名字的function，并且抛出一个语法错误信息，因为function声明需要一个名字。

旁白：函数(function)，括弧(paren)，语法错误(SyntaxError)

有趣的是，即便你为上面那个错误的代码加上一个名字，他也会提示语法错误，只不过和上面的原因不一样。在一个表达式后面加上括号()，该表达式会立即执行，但是在一个语句后面加上括号()，是完全不一样的意思，他的只是分组操作符。

```
// 下面这个function在语法上是没问题的，但是依然只是一个语句
// 加上括号()以后依然会报错，因为分组操作符需要包含表达式

function foo(){ /* code */ }(); // SyntaxError: Unexpected token )

// 但是如果你在括弧()里传入一个表达式，将不会有异常抛出
// 但是foo函数依然不会执行
function foo(){ /* code */ }( 1 );

// 因为它完全等价于下面这个代码，一个function声明后面，又声明了一个毫无关系的表达式：

function foo(){ /* code */ }

( 1 );
```

你可以访问[ECMA-262-3 in detail. Chapter 5. Functions](#) 获取进一步的信息。

自执行函数表达式

要解决上述问题，非常简单，我们只需要用大括弧将代码的代码全部括住就行了，因为JavaScript里括弧()里面不能包含语句，所以在这一点上，解析器在解析function关键字的时候，会将相应的代码解析成function表达式，而不是function声明。

```
// 下面2个括弧()都会立即执行

(function () { /* code */ } ()); // 推荐使用这个
(function () { /* code */ })(); // 但是这个也是可以用的

// 由于括弧()和JS的&&, 异或, 逗号等操作符是在函数表达式和函数声明上消除歧义的
// 所以一旦解析器知道其中一个已经是表达式了, 其它的也都默认为表达式了
// 不过, 请注意下一章节的内容解释

var i = function () { return 10; } ();
true && function () { /* code */ } ();
0, function () { /* code */ } ();

// 如果你不在意返回值, 或者不怕难以阅读
// 你甚至可以在function前面加一元操作符号

!function () { /* code */ } ();
~function () { /* code */ } ();
-function () { /* code */ } ();
+function () { /* code */ } ();

// 还有一个情况, 使用new关键字, 也可以用, 但我不确定它的效率
// http://twitter.com/kuvos/status/18209252090847232

new function () { /* code */ }
new function () { /* code */ } () // 如果需要传递参数, 只需要加上括弧()
```

上面所说的括弧是消除歧义的，其实压根就没必要，因为括弧本来内部本来期望的就是函数表达式，但是我们依然用它，主要是为了方便开发人员阅读，当你让这些已经自动执行的表达式赋值给一个变量的时候，我们看到开头有括弧(，很快就能明白，而不需要将代码拉到最后看看到底有没有加括弧。

用闭包保存状态

和普通function执行的时候传参数一样，自执行的函数表达式也可以这么传参，因为闭包直接可以引用传入的这些参数，利用这些被lock住的传入参数，自执行函数表达式可以有效地保存状态。

```
// 这个代码是错误的, 因为变量i从来就没被locked住
// 相反, 当循环执行以后, 我们在点击的时候i才获得数值
// 因为这个时候i才真正获得值
// 所以说无论点击那个连接, 最终显示的都是I am link #10 (如果有10个a元素的话)
```

本文档使用 [看云](#) 构建

```

var elems = document.getElementsByTagName('a');

for (var i = 0; i < elems.length; i++) {
    elems[i].addEventListener('click', function (e) {
        e.preventDefault();
        alert('I am link #' + i);
    }, 'false');
}

// 这个是可以用的，因为他在自执行函数表达式闭包内部
// i的值作为locked的索引存在，在循环执行结束以后，尽管最后i的值变成了a元素总数（例如10）
// 但闭包内部的lockedInIndex值是没有改变，因为他已经执行完毕了
// 所以当点击连接的时候，结果是正确的

var elems = document.getElementsByTagName('a');

for (var i = 0; i < elems.length; i++) {
    (function (lockedInIndex) {
        elems[i].addEventListener('click', function (e) {
            e.preventDefault();
            alert('I am link #' + lockedInIndex);
        }, 'false');
    })(i);
}

// 你也可以像下面这样应用，在处理函数那里使用自执行函数表达式
// 而不是在addEventListener外部
// 但是相对来说，上面的代码更具可读性

var elems = document.getElementsByTagName('a');

for (var i = 0; i < elems.length; i++) {
    elems[i].addEventListener('click', (function (lockedInIndex) {
        return function (e) {
            e.preventDefault();
            alert('I am link #' + lockedInIndex);
        };
    })(i), 'false');
}

```

其实，上面2个例子中的lockedInIndex变量，也可以换成i，因为和外面的i不在一个作用于，所以不会出现问题，这也是匿名函数+闭包的威力。

自执行匿名函数和立即执行的函数表达式区别

在这篇帖子里，我们一直叫自执行函数，确切的说是自执行匿名函数（Self-executing anonymous function），但英文原文作者一直倡议使用立即调用的函数表达式（Immediately-Invoked Function Expression）这一名称，作者又举了一堆例子来解释，好吧，我们来看看：

```
// 这是一个自执行的函数，函数内部执行自身，递归
function foo() { foo(); }

// 这是一个自执行的匿名函数，因为没有标示名称
// 必须使用arguments.callee属性来执行自己
var foo = function () { arguments.callee(); };

// 这可能也是一个自执行的匿名函数，仅仅是foo标示名称引用它自身
// 如果你将foo改变成其它的，你将得到一个used-to-self-execute匿名函数
var foo = function () { foo(); };

// 有些人叫这个是自执行的匿名函数（即便它不是），因为它没有调用自身，它只是立即执行而已。
(function () { /* code */ } ());

// 为函数表达式添加一个标示名称，可以方便Debug
// 但一定命名了，这个函数就不再是匿名的了
(function foo() { /* code */ } ());

// 立即调用的函数表达式（IIFE）也可以自执行，不过可能不常用罢了
(function () { arguments.callee(); } ());
(function foo() { foo(); } ());

// 另外，下面的代码在黑莓5里执行会出错，因为在一个命名的函数表达式里，他的名称是undefined
// 呵呵，奇怪
(function foo() { foo(); } ());
```

希望这里的一些例子，可以让大家明白，什么叫自执行，什么叫立即调用。

注：arguments.callee在ECMAScript 5 strict mode里被废弃了，所以在这个模式下，其实是不能用的。

最后的旁白：Module模式

在讲到这个立即调用的函数表达式的时候，我又想起来了Module模式，如果你还不熟悉这个模式，我们先来看看代码：

```
// 创建一个立即调用的匿名函数表达式
// return一个变量，其中这个变量里包含你要暴露的东西
// 返回的这个变量将赋值给counter，而不是外面声明的function自身
```

```
var counter = (function () {  
    var i = 0;  
  
    return {  
        get: function () {  
            return i;  
        },  
        set: function (val) {  
            i = val;  
        },  
        increment: function () {  
            return ++i;  
        }  
    };  
})();
```

// counter是一个带有多个属性的对象，上面的代码对于属性的体现其实是方法

```
counter.get(); // 0  
counter.set(3);  
counter.increment(); // 4  
counter.increment(); // 5
```

```
counter.i; // undefined 因为i不是返回对象的属性  
i; // 引用错误: i 没有定义 (因为i只存在于闭包)
```

关于更多Module模式的介绍，请访问我的上一篇帖子：深入理解JavaScript系列（2）：全面解析Module模式。

更多阅读

希望上面的一些例子，能让你对立即调用的函数表达（也就是我们所说的自执行函数）有所了解，如果你想了解更多关于function和Module模式的信息，请继续访问下面列出的网站：

1. [ECMA-262-3 in detail. Chapter 5. Functions.](#) - Dmitry A. Soshnikov
2. [Functions and function scope](#) - Mozilla Developer Network
3. [Named function expressions](#) - Juriy “kangax” Zaytsev
4. [全面解析Module模式](#)- Ben Cherry (大叔翻译整理)
5. [Closures explained with JavaScript](#) - Nick Morgan

(5) 强大的原型和原型链

前言

JavaScript 不包含传统的类继承模型，而是使用 prototypal 原型模型。

虽然这经常被当作是 JavaScript 的缺点被提及，其实基于原型的继承模型比传统的类继承还要强大。实现传统的类继承模型是很简单，但是实现 JavaScript 中的原型继承则要困难的多。

由于 JavaScript 是唯一一个被广泛使用的基于原型继承的语言，所以理解两种继承模式的差异是需要一定时间的，今天我们就来了解一下原型和原型链。

原型

10年前，我刚学习JavaScript的时候，一般都是用如下方式来写代码：

```
var decimalDigits = 2,
    tax = 5;

function add(x, y) {
    return x + y;
}

function subtract(x, y) {
    return x - y;
}

//alert(add(1, 3));
```

通过执行各个function来得到结果，学习了原型之后，我们可以使用如下方式来美化一下代码。

原型使用方式1：

在使用原型之前，我们需要先将代码做一下小修改：

```
var Calculator = function (decimalDigits, tax) {
    this.decimalDigits = decimalDigits;
    this.tax = tax;
};
```

然后，通过给Calculator对象的prototype属性赋值对象字面量来设定Calculator对象的原

型。

```
Calculator.prototype = {
  add: function (x, y) {
    return x + y;
  },
  subtract: function (x, y) {
    return x - y;
  }
};
//alert((new Calculator()).add(1, 3));
```

这样，我们就可以new Calculator对象以后，就可以调用add方法来计算结果了。

原型使用方式2：

第二种方式是，在赋值原型prototype的时候使用function立即执行的表达式来赋值，即如下格式：

```
Calculator.prototype = function () { } ();
```

它的好处在前面的帖子里已经知道了，就是可以封装私有的function，通过return的形式暴露出简单的使用名称，以达到public/private的效果，修改后的代码如下：

```
Calculator.prototype = function () {
  add = function (x, y) {
    return x + y;
  },
  subtract = function (x, y) {
    return x - y;
  }
  return {
    add: add,
    subtract: subtract
  }
} ();

//alert((new Calculator()).add(11, 3));
```

同样的方式，我们可以new Calculator对象以后调用add方法来计算结果了。

再来一点

分步声明：

上述使用原型的时候，有一个限制就是一次性设置了原型对象，我们再来说一下如何分来设置原型的每个属性吧。

```
var BaseCalculator = function () {
    //为每个实例都声明一个小数位数
    this.decimalDigits = 2;
};

//使用原型给BaseCalculator扩展2个对象方法
BaseCalculator.prototype.add = function (x, y) {
    return x + y;
};

BaseCalculator.prototype.subtract = function (x, y) {
    return x - y;
};
```

首先，声明了一个BaseCalculator对象，构造函数里会初始化一个小数位数的属性 decimalDigits，然后通过原型属性设置2个function，分别是add(x,y)和subtract(x,y)，当然你也可以使用前面提到的2种方式的任何一种，我们的主要目的是看如何将BaseCalculator对象设置到真正的Calculator的原型上。

```
var BaseCalculator = function() {
    this.decimalDigits = 2;
};

BaseCalculator.prototype = {
    add: function(x, y) {
        return x + y;
    },
    subtract: function(x, y) {
        return x - y;
    }
};
```

创建完上述代码以后，我们来开始：

```
var Calculator = function () {
    //为每个实例都声明一个税收数字
    this.tax = 5;
};

Calculator.prototype = new BaseCalculator();
```

我们可以看到Calculator的原型是指向到BaseCalculator的一个实例上，目的是让Calculator

集成它的add(x,y)和subtract(x,y)这2个function，还有一点要说的是，由于它的原型是BaseCalculator的一个实例，所以不管你创建多少个Calculator对象实例，他们的原型指向的都是同一个实例。

```
var calc = new Calculator();
alert(calc.add(1, 1));
//BaseCalculator 里声明的decimalDigits属性, 在 Calculator里是可以访问到的
alert(calc.decimalDigits);
```

上面的代码，运行以后，我们可以看到因为Calculator的原型是指向BaseCalculator的实例上的，所以可以访问他的decimalDigits属性值，那如果我不想让Calculator访问BaseCalculator的构造函数里声明的属性值，那怎么办呢？这么办：

```
var Calculator = function () {
    this.tax= 5;
};

Calculator.prototype = BaseCalculator.prototype;
```

通过将BaseCalculator的原型赋给Calculator的原型，这样你在Calculator的实例上就访问不到那个decimalDigits值了，如果你访问如下代码，那将会提升出错。

```
var calc = new Calculator();
alert(calc.add(1, 1));
alert(calc.decimalDigits);
```

重写原型：

在使用第三方JS类库的时候，往往有时候他们定义的原型方法是不能满足我们的需要，但是又离不开这个类库，所以这时候我们就需要重写他们的原型中的一个或者多个属性或function，我们可以通过继续声明的同样的add代码的形式来达到覆盖重写前面的add功能，代码如下：

```
//覆盖前面Calculator的add() function
Calculator.prototype.add = function (x, y) {
    return x + y + this.tax;
};

var calc = new Calculator();
alert(calc.add(1, 1));
```

这样，我们计算得出的结果就比原来多出了一个tax的值，但是有一点需要注意：那就是重写的代码需要放在最后，这样才能覆盖前面的代码。

原型链

在讲原型链之前，我们先上一段代码：

```
function Foo() {
    this.value = 42;
}
Foo.prototype = {
    method: function() {}
};

function Bar() {}

// 设置Bar的prototype属性为Foo的实例对象
Bar.prototype = new Foo();
Bar.prototype.foo = 'Hello World';

// 修正Bar.prototype.constructor为Bar本身
Bar.prototype.constructor = Bar;

var test = new Bar() // 创建Bar的一个新实例

// 原型链
test [Bar的实例]
    Bar.prototype [Foo的实例]
        { foo: 'Hello World' }
    Foo.prototype
        {method: ...};
    Object.prototype
        {toString: ... /* etc. */};
```

上面的例子中，test 对象从 Bar.prototype 和 Foo.prototype 继承下来；因此，它能访问 Foo 的原型方法 method。同时，它也能够访问那个定义在原型上的 Foo 实例属性 value。需要注意的是 new Bar() 不会创造出一个新的 Foo 实例，而是重复使用它原型上的那个实例；因此，所有的 Bar 实例都会共享相同的 value 属性。

属性查找：

当查找一个对象的属性时，JavaScript 会向上遍历原型链，直到找到给定名称的属性为止，到查找到达原型链的顶部 - 也就是 Object.prototype - 但是仍然没有找到指定的属性，就会返回 undefined，我们来看一个例子：

```
function foo() {
    this.add = function (x, y) {
        return x + y;
    };
}
```

```

    }
}

foo.prototype.add = function (x, y) {
    return x + y + 10;
}

Object.prototype.subtract = function (x, y) {
    return x - y;
}

var f = new foo();
alert(f.add(1, 2)); //结果是3, 而不是13
alert(f.subtract(1, 2)); //结果是-1

```

通过代码运行，我们发现subtract是安装我们所说的向上查找来得到结果的，但是add方式有点小不同，这也是我想强调的，就是属性在查找的时候是先查找自身的属性，如果没有再查找原型，再没有，再往上走，一直插到Object的原型上，所以在某种层面上说，用 for in 语句遍历属性时，效率也是个问题。

还有一点我们需要注意的是，我们可以赋值任何类型的对象到原型上，但是不能赋值原子类型的值，比如如下代码是无效的：

```

function Foo() {}
Foo.prototype = 1; // 无效

```

hasOwnProperty函数：

hasOwnProperty是Object.prototype的一个方法，它可是个好东西，他能判断一个对象是否包含自定义属性而不是原型链上的属性，因为hasOwnProperty 是 JavaScript 中唯一一个处理属性但是不查找原型链的函数。

```

// 修改Object.prototype
Object.prototype.bar = 1;
var foo = {goo: undefined};

foo.bar; // 1
'bar' in foo; // true

foo.hasOwnProperty('bar'); // false
foo.hasOwnProperty('goo'); // true

```

只有 hasOwnProperty 可以给出正确和期望的结果，这在遍历对象的属性时会很有用。没有其它方法可以用来排除原型链上的属性，而不是定义在对象自身上的属性。

但有个恶心的地方是：JavaScript 不会保护 `hasOwnProperty` 被非法占用，因此如果一个对象碰巧存在这个属性，就需要使用外部的 `hasOwnProperty` 函数来获取正确的结果。

```
var foo = {
  hasOwnProperty: function() {
    return false;
  },
  bar: 'Here be dragons'
};

foo.hasOwnProperty('bar'); // 总是返回 false

// 使用{}对象的 hasOwnProperty, 并将其上下为设置为foo
{}.hasOwnProperty.call(foo, 'bar'); // true
```

当检查对象上某个属性是否存在时，`hasOwnProperty` 是唯一可用的方法。同时在使用 `for in` loop 遍历对象时，推荐总是使用 `hasOwnProperty` 方法，这将会避免原型对象扩展带来的干扰，我们来看一下例子：

```
// 修改 Object.prototype
Object.prototype.bar = 1;

var foo = {moo: 2};
for(var i in foo) {
  console.log(i); // 输出两个属性：bar 和 moo
}
```

我们没办法改变 `for in` 语句的行为，所以想过滤结果就只能使用 `hasOwnProperty` 方法，代码如下：

```
// foo 变量是上例中的
for(var i in foo) {
  if (foo.hasOwnProperty(i)) {
    console.log(i);
  }
}
```

这个版本的代码是唯一正确的写法。由于我们使用了 `hasOwnProperty`，所以这次只输出 `moo`。如果不使用 `hasOwnProperty`，则这段代码在原生对象原型（比如 `Object.prototype`）被扩展时可能会出错。

总结：推荐使用 `hasOwnProperty`，不要对代码运行的环境做任何假设，不要假设原生对象是否已经被扩展了。

总结

原型极大地丰富了我们的开发代码，但是在平时使用的过程中一定要注意上述提到的一些注意事项。

参考内容：<http://bonsaiden.github.com/JavaScript-Garden/zh/>

(6) S.O.L.I.D五大原则之单一职责SRP

前言

Bob大叔提出并发扬了S.O.L.I.D五大原则，用来更好地进行面向对象编程，五大原则分别是：

1. The Single Responsibility Principle (单一职责SRP)
2. The Open/Closed Principle (开闭原则OCP)
3. The Liskov Substitution Principle (里氏替换原则LSP)
4. The Interface Segregation Principle (接口分离原则ISP)
5. The Dependency Inversion Principle (依赖反转原则DIP)

五大原则，我相信在博客园已经被讨论烂了，尤其是C#的实现，但是相对于JavaScript这种以原型为base的动态类型语言来说还为数不多，该系列将分5篇文章以JavaScript编程语言为基础来展示五大原则的应用。 OK，开始我们的第一篇：单一职责。

英文原文：<http://freshbrewedcode.com/derekgreer/2011/12/08/solid-javascript-single-responsibility-principle/>

单一职责

单一职责的描述如下：

A class should have only one reason to change
类发生更改的原因应该只有一个

一个类（JavaScript下应该是一个对象）应该有一组紧密相关的行为的意思是什么？遵守单一职责的好处是可以帮助我们很容易地来维护这个对象，当一个对象封装了很多职责的话，一旦一个职责需要修改，势必会影响该对象想的其它职责代码。通过解耦可以让每个职责工更加有弹性地变化。

不过，我们如何知道一个对象的多个行为构造多个职责还是单个职责？我们可以通过参考[Object Design: Roles, Responsibilities, and Collaborations](#)一书提出的Role Stereotypes概念来决定，该书提出了如下Role Stereotypes来区分职责：

1. Information holder – 该对象设计为存储对象并提供对象信息给其它对象。
2. Structurer – 该对象设计为维护对象和信息之间的关系

3. Service provider – 该对象设计为处理工作并提供服务给其它对象
4. Controller – 该对象设计为控制决策一系列负责的任务处理
5. Coordinator – 该对象不做任何决策处理工作，只是delegate工作到其它对象上
6. Interfacer – 该对象设计为在系统的各个部分转化信息（或请求）

一旦你知道了这些概念，那就很容易知道你的代码到底是多职责还是单一职责了。

实例代码

该实例代码演示的是将商品添加到购物车，代码非常糟糕，代码如下：

```
function Product(id, description) {
    this.getId = function () {
        return id;
    };
    this.getDescription = function () {
        return description;
    };
}

function Cart(eventAggregator) {
    var items = [];

    this.addItem = function (item) {
        items.push(item);
    };
}

(function () {
    var products = [new Product(1, "Star Wars Lego Ship"),
                    new Product(2, "Barbie Doll"),
                    new Product(3, "Remote Control Airplane")],
        cart = new Cart();

    function addToCart() {
        var productId = $(this).attr('id');
        var product = $.grep(products, function (x) {
            return x.getId() == productId;
        })[0];
        cart.addItem(product);

        var newItem = $('<li></li>').html(product.getDescription()).attr(
            'id-cart', product.getId()).appendTo("#cart");
    }

    products.forEach(function (product) {
        var newItem = $('<li></li>').html(product.getDescription())
            .attr('id', product.getId())
            .dblclick(addToCart)
            .appendTo("#products");
    });
})();
```

该代码声明了2个function分别用来描述product和cart，而匿名函数的职责是更新屏幕和用户交互，这还不是一个很复杂的例子，但匿名函数里却包含了很多不相关的职责，让我们来看看到底有多少职责：

1. 首先，有product的集合的声明
2. 其次，有一个将product集合绑定到#product元素的代码，而且还附件了一个添加到购物车的事件处理
3. 第三，有Cart购物车的展示功能
4. 第四，有添加product item到购物车并显示的功能

重构代码

让我们来分解一下，以便代码各自存放到各自的对象里，为此，我们参考了martinfowler的事件聚合（[Event Aggregator](#)）理论在处理代码以便各对象之间进行通信。

首先我们先来实现事件聚合的功能，该功能分为2部分，1个是Event，用于Handler回调的代码，1个是EventAggregator用来订阅和发布Event，代码如下：

```
function Event(name) {
    var handlers = [];

    this.getName = function () {
        return name;
    };

    this.addHandler = function (handler) {
        handlers.push(handler);
    };

    this.removeHandler = function (handler) {
        for (var i = 0; i < handlers.length; i++) {
            if (handlers[i] == handler) {
                handlers.splice(i, 1);
                break;
            }
        }
    };

    this.fire = function (eventArgs) {
        handlers.forEach(function (h) {
            h(eventArgs);
        });
    };
}

function EventAggregator() {
```

```

var events = [];

function getEvent(eventName) {
    return $.grep(events, function (event) {
        return event.getName() === eventName;
    })[0];
}

this.publish = function (eventName, eventArgs) {
    var event = getEvent(eventName);

    if (!event) {
        event = new Event(eventName);
        events.push(event);
    }
    event.fire(eventArgs);
};

this.subscribe = function (eventName, handler) {
    var event = getEvent(eventName);

    if (!event) {
        event = new Event(eventName);
        events.push(event);
    }

    event.addHandler(handler);
};
}

```

然后，我们来声明Product对象，代码如下：

```

function Product(id, description) {
    this.getId = function () {
        return id;
    };
    this.getDescription = function () {
        return description;
    };
}

```

接着来声明Cart对象，该对象的addItem的function里我们要触发发布一个事件itemAdded，然后将item作为参数传进去。

```

function Cart(eventAggregator) {
    var items = [];

    this.addItem = function (item) {
        items.push(item);
        eventAggregator.publish("itemAdded", item);
    };
}

```

```
    };
  }
```

CartController主要是接受cart对象和事件聚合器，通过订阅itemAdded来增加一个li元素节点，通过订阅productSelected事件来添加product。

```
function CartController(cart, eventAggregator) {
  eventAggregator.subscribe("itemAdded", function (eventArgs) {
    var newItem = $('<li></li>').html(eventArgs.getDescription()).attr('id-cart', eventArgs.getId()).appendTo("#cart");
  });

  eventAggregator.subscribe("productSelected", function (eventArgs) {
    cart.addItem(eventArgs.product);
  });
}
```

Repository的目的是为了获取数据（可以从ajax里获取），然后暴露get数据的方法。

```
function ProductRepository() {
  var products = [new Product(1, "Star Wars Lego Ship"),
    new Product(2, "Barbie Doll"),
    new Product(3, "Remote Control Airplane")];

  this.getProducts = function () {
    return products;
  }
}
```

ProductController里定义了一个onProductSelect方法，主要是发布触发productSelected事件，forEach主要是用于绑定数据到产品列表上，代码如下：

```
function ProductController(eventAggregator, productRepository) {
  var products = productRepository.getProducts();

  function onProductSelected() {
    var productId = $(this).attr('id');
    var product = $.grep(products, function (x) {
      return x.getId() == productId;
    })[0];
    eventAggregator.publish("productSelected", {
      product: product
    });
  }

  products.forEach(function (product) {
    var newItem = $('<li></li>').html(product.getDescription())
```

```

        .attr('id', product.getId())
        .dblclick(onProductSelected)
        .appendTo("#products");
    });
}

```

最后声明匿名函数（需要确保HTML都加载完了才能执行这段代码，比如放在jQuery的ready方法里）：

```

(function () {
    var eventAggregator = new EventAggregator(),
        cart = new Cart(eventAggregator),
        cartController = new CartController(cart, eventAggregator),
        productRepository = new ProductRepository(),
        productController = new ProductController(eventAggregator, productRepository);
})();

```

可以看到匿名函数的代码减少了很多，主要是一个对象的实例化代码，代码里我们介绍了Controller的概念，他接受信息然后传递到action，我们也介绍了Repository的概念，主要是用来处理product的展示，重构的结果就是写了一大堆的对象声明，但是好处是每个对象有了自己明确的职责，该展示数据的展示数据，改处理集合的处理集合，这样耦合度就非常低了。

```

function Event(name) {
    var handlers = [];

    this.getName = function () {
        return name;
    };

    this.addHandler = function (handler) {
        handlers.push(handler);
    };

    this.removeHandler = function (handler) {
        for (var i = 0; i < handlers.length; i++) {
            if (handlers[i] == handler) {
                handlers.splice(i, 1);
                break;
            }
        }
    };

    this.fire = function (eventArgs) {
        handlers.forEach(function (h) {
            h(eventArgs);
        });
    };
}

```

```

    };
}

function EventAggregator() {
    var events = [];

    function getEvent(eventName) {
        return $.grep(events, function (event) {
            return event.getName() === eventName;
        })[0];
    }

    this.publish = function (eventName, eventArgs) {
        var event = getEvent(eventName);

        if (!event) {
            event = new Event(eventName);
            events.push(event);
        }
        event.fire(eventArgs);
    };

    this.subscribe = function (eventName, handler) {
        var event = getEvent(eventName);

        if (!event) {
            event = new Event(eventName);
            events.push(event);
        }

        event.addHandler(handler);
    };
}

function Product(id, description) {
    this.getId = function () {
        return id;
    };
    this.getDescription = function () {
        return description;
    };
}

function Cart(eventAggregator) {
    var items = [];

    this.addItem = function (item) {
        items.push(item);
        eventAggregator.publish("itemAdded", item);
    };
}

function CartController(cart, eventAggregator) {
    eventAggregator.subscribe("itemAdded", function (eventArgs) {
        var newItem = $('<li></li>').html(eventArgs.getDescription()).attr('id-cart', eventArgs.getId()).appendTo("#cart");
    });
}

```

```

    });

    eventAggregator.subscribe("productSelected", function (eventArgs) {
        cart.addItem(eventArgs.product);
    });
}

function ProductRepository() {
    var products = [new Product(1, "Star Wars Lego Ship"),
        new Product(2, "Barbie Doll"),
        new Product(3, "Remote Control Airplane")];

    this.getProducts = function () {
        return products;
    }
}

function ProductController(eventAggregator, productRepository) {
    var products = productRepository.getProducts();

    function onProductSelected() {
        var productId = $(this).attr('id');
        var product = $.grep(products, function (x) {
            return x.getId() == productId;
        })[0];
        eventAggregator.publish("productSelected", {
            product: product
        });
    }

    products.forEach(function (product) {
        var newItem = $('<li></li>').html(product.getDescription())
            .attr('id', product.getId())
            .dblclick(onProductSelected)
            .appendTo("#products");
    });
}

(function () {
    var eventAggregator = new EventAggregator(),
        cart = new Cart(eventAggregator),
        cartController = new CartController(cart, eventAggregator),
        productRepository = new ProductRepository(),
        productController = new ProductController(eventAggregator, productRepository);
})();

```

总结

看到这个重构结果，有博友可能要问了，真的有必要做这么复杂么？我只能说：要不要这么做取决于你项目的情况。

如果你的项目是个是个非常小的项目，代码也不是很多，那其实是没有必要重构得这么复

杂，但如果你的项目是个很复杂的大型项目，或者你的小项目将来可能增长得很快快的话，那就在前期就得考虑SRP原则进行职责分离了，这样才有利于以后的维护。

(7) S.O.L.I.D五大原则之开闭原则OCP

前言

本章我们要讲解的是S.O.L.I.D五大原则JavaScript语言实现的第2篇，开闭原则OCP（The Open/Closed Principle）。

开闭原则的描述是：

Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.

软件实体（类，模块，方法等等）应当对扩展开放，对修改关闭，即软件实体应当在不修改的前提下扩展。

open for extension（对扩展开放）的意思是说当新需求出现的时候，可以通过扩展现有模型达到目的。而Close for modification（对修改关闭）的意思是不允许对该实体做任何修改，说白了，就是这些需要执行多样行为的实体应该设计成不需要修改就可以实现各种的变化，坚持开闭原则有利于用最少的代码进行项目维护。

英文原文：<http://freshbrewedcode.com/derekgreer/2011/12/19/solid-javascript-the-openclosed-principle/>

问题代码

为了直观地描述，我们来举个例子演示一下，下属代码是动态展示question列表的代码（没有使用开闭原则）。

```
// 问题类型
var AnswerType = {
    Choice: 0,
    Input: 1
};

// 问题实体
function question(label, answerType, choices) {
    return {
        label: label,
        answerType: answerType,
        choices: choices // 这里的choices是可选参数
    };
}
```

```

var view = (function () {
    // render一个问题
    function renderQuestion(target, question) {
        var questionWrapper = document.createElement('div');
        questionWrapper.className = 'question';

        var questionLabel = document.createElement('div');
        questionLabel.className = 'question-label';
        var label = document.createTextNode(question.label);
        questionLabel.appendChild(label);

        var answer = document.createElement('div');
        answer.className = 'question-input';

        // 根据不同的类型展示不同的代码：分别是下拉菜单和输入框两种
        if (question.answerType === AnswerType.Choice) {
            var input = document.createElement('select');
            var len = question.choices.length;
            for (var i = 0; i < len; i++) {
                var option = document.createElement('option');
                option.text = question.choices[i];
                option.value = question.choices[i];
                input.appendChild(option);
            }
        }
        else if (question.answerType === AnswerType.Input) {
            var input = document.createElement('input');
            input.type = 'text';
        }

        answer.appendChild(input);
        questionWrapper.appendChild(questionLabel);
        questionWrapper.appendChild(answer);
        target.appendChild(questionWrapper);
    }

    return {
        // 遍历所有的问题列表进行展示
        render: function (target, questions) {
            for (var i = 0; i < questions.length; i++) {
                renderQuestion(target, questions[i]);
            }
        }
    };
})();

var questions = [
    question('Have you used tobacco products within the last
30 days?', AnswerType.Choice, ['Yes', 'No']),
    question('What medications are you currently using?', Ans
werType.Input)
];

var questionRegion = document.getElementById('questions');
view.render(questionRegion, questions);

```

上面的代码，view对象里包含一个render方法用来展示question列表，展示的时候根据不同的question类型使用不同的展示方式，一个question包含一个label和一个问题类型以及choices的选项（如果是选择类型的话）。如果问题类型是Choice那就根据选项生产一个下拉菜单，如果类型是Input，那就简单地展示input输入框。

该代码有一个限制，就是如果再增加一个question类型的话，那就需要再次修改renderQuestion里的条件语句，这明显违反了开闭原则。

重构代码

让我们来重构一下这个代码，以便在出现新question类型的情况下允许扩展view对象的render能力，而不需要修改view对象内部的代码。

先来创建一个通用的questionCreator函数：

```
function questionCreator(spec, my) {
  var that = {};

  my = my || {};
  my.label = spec.label;

  my.renderInput = function () {
    throw "not implemented";
    // 这里renderInput没有实现，主要目的是让各自问题类型的实现代码去覆盖整个方法
  };

  that.render = function (target) {
    var questionWrapper = document.createElement('div');
    questionWrapper.className = 'question';

    var questionLabel = document.createElement('div');
    questionLabel.className = 'question-label';
    var label = document.createTextNode(spec.label);
    questionLabel.appendChild(label);

    var answer = my.renderInput();
    // 该render方法是同样的粗合理代码
    // 唯一的不同就是上面的一句my.renderInput()
    // 因为不同的问题类型有不同的实现

    questionWrapper.appendChild(questionLabel);
    questionWrapper.appendChild(answer);
    return questionWrapper;
  };

  return that;
}
```

该代码的作用组合要是render一个问题，同时提供一个未实现的renderInput方法以便其他function可以覆盖，以使用不同的问题类型，我们继续看一下每个问题类型的实现代码：

```
function choiceQuestionCreator(spec) {
    var my = {},
    that = questionCreator(spec, my);

    // choice类型的renderInput实现
    my.renderInput = function () {
        var input = document.createElement('select');
        var len = spec.choices.length;
        for (var i = 0; i < len; i++) {
            var option = document.createElement('option');
            option.text = spec.choices[i];
            option.value = spec.choices[i];
            input.appendChild(option);
        }

        return input;
    };

    return that;
}

function inputQuestionCreator(spec) {
    var my = {},
    that = questionCreator(spec, my);

    // input类型的renderInput实现
    my.renderInput = function () {
        var input = document.createElement('input');
        input.type = 'text';
        return input;
    };

    return that;
}
```

choiceQuestionCreator函数和inputQuestionCreator函数分别对应下拉菜单和input输入框的renderInput实现，通过内部调用统一的questionCreator(spec, my)然后返回that对象（同一类型哦）。

view对象的代码就很固定了。

```
var view = {
    render: function(target, questions) {
        for (var i = 0; i < questions.length; i++) {
            target.appendChild(questions[i].render());
        }
    }
}
```

```

    }
  }
};

```

所以我们声明问题的时候只需要这样做，就OK了：

```

var questions = [
  choiceQuestionCreator({
    label: 'Have you used tobacco products within the last 30 days?',
    choices: ['Yes', 'No']
  }),
  inputQuestionCreator({
    label: 'What medications are you currently using?'
  })
];

```

最终的使用代码，我们可以这样来用：

```

var questionRegion = document.getElementById('questions');

view.render(questionRegion, questions);

```

最后重构的代码如下：

```

function questionCreator(spec, my) {
  var that = {};

  my = my || {};
  my.label = spec.label;

  my.renderInput = function() {
    throw "not implemented";
  };

  that.render = function(target) {
    var questionWrapper = document.createElement('div');
    questionWrapper.className = 'question';

    var questionLabel = document.createElement('div');
    questionLabel.className = 'question-label';
    var label = document.createTextNode(spec.label);
    questionLabel.appendChild(label);

    var answer = my.renderInput();

    questionWrapper.appendChild(questionLabel);
    questionWrapper.appendChild(answer);
    return questionWrapper;
  };
};

```

```

        return that;
    }

    function choiceQuestionCreator(spec) {

        var my = {},
            that = questionCreator(spec, my);

        my.renderInput = function() {
            var input = document.createElement('select');
            var len = spec.choices.length;
            for (var i = 0; i < len; i++) {
                var option = document.createElement('option');
                option.text = spec.choices[i];
                option.value = spec.choices[i];
                input.appendChild(option);
            }

            return input;
        };

        return that;
    }

    function inputQuestionCreator(spec) {

        var my = {},
            that = questionCreator(spec, my);

        my.renderInput = function() {
            var input = document.createElement('input');
            input.type = 'text';
            return input;
        };

        return that;
    }

    var view = {
        render: function(target, questions) {
            for (var i = 0; i < questions.length; i++) {
                target.appendChild(questions[i].render());
            }
        }
    };

    var questions = [
        choiceQuestionCreator({
            label: 'Have you used tobacco products within the last 30 days?',
            choices: ['Yes', 'No']
        }),
        inputQuestionCreator({
            label: 'What medications are you currently using?'
        })
    ];

```

```
var questionRegion = document.getElementById('questions');  
view.render(questionRegion, questions);
```

上面的代码里应用了一些技术点，我们来逐一看一下：

1. 首先，questionCreator方法的创建，可以让我们使用[模板方法模式](#)将处理问题的功能delegat给针对每个问题类型的扩展代码renderInput上。
2. 其次，我们用一个私有的spec属性替换掉了前面question方法的构造函数属性，因为我们封装了render行为进行操作，不再需要把这些属性暴露给外部代码了。
3. 第三，我们为每个问题类型创建一个对象进行各自的代码实现，但每个实现里都必须包含renderInput方法以便覆盖questionCreator方法里的renderInput代码，这就是我们常说的[策略模式](#)。

通过重构，我们可以去除不必要的问题类型的枚举AnswerType，而且可以让choices作为choiceQuestionCreator函数的必选参数（之前的版本是一个可选参数）。

总结

重构以后的版本的view对象可以很清晰地进行新的扩展了，为不同的问题类型扩展新的对象，然后声明questions集合的时候再里面指定类型就行了，view对象本身不再修改任何改变，从而达到了开闭原则的要求。

另：懂C#的话，不知道看了上面的代码后是否和多态的实现有些类似？其实上述的代码用原型也是可以实现的，大家可以自行研究一下。

(8) S.O.L.I.D五大原则之里氏替换原则LSP

前言

本章我们要讲解的是S.O.L.I.D五大原则JavaScript语言实现的第3篇，里氏替换原则LSP (The Liskov Substitution Principle)。

英文原文：<http://freshbrewedcode.com/derekgreer/2011/12/31/solid-javascript-the-liskov-substitution-principle/>

开闭原则的描述是：

Subtypes must be substitutable for their base types.

派生类型必须可以替换它的基类型。

在面向对象编程里，继承提供了一个机制让子类和共享基类的代码，这是通过在基类型里封装通用的数据和行为来实现的，然后已经及类型来声明更详细的子类型，为了应用里氏替换原则，继承子类型需要在语义上等价于基类型里的期望行为。

为了来更好的理解，请参考如下代码：

```
function Vehicle(my) {
  var my = my || {};
  my.speed = 0;
  my.running = false;

  this.speed = function() {
    return my.speed;
  };
  this.start = function() {
    my.running = true;
  };
  this.stop = function() {
    my.running = false;
  };
  this.accelerate = function() {
    my.speed++;
  };
  this.decelerate = function() {
    my.speed--;
  }, this.state = function() {
    if (!my.running) {
      return "parked";
    }
  }
}
```

```

        else if (my.running && my.speed) {
            return "moving";
        }
        else if (my.running) {
            return "idle";
        }
    };
}

```

上述代码我们定义了一个Vehicle函数，其构造函数为vehicle对象提供了一些基本的操作，我们来想想如果当前函数当前正运行在服务客户的产品环境上，如果现在需要添加一个新的构造函数来实现加快移动的vehicle。思考以后，我们写出了如下代码：

```

function FastVehicle(my) {
    var my = my || {};

    var that = new Vehicle(my);
    that.accelerate = function() {
        my.speed += 3;
    };
    return that;
}

```

在浏览器的控制台我们都测试了，所有的功能都是我们的预期，没有问题，FastVehicle的速度增快了3倍，而且继承他的方法也是按照我们的预期工作。此后，我们开始部署这个新版本的类库到产品环境上，可是我们却接到了新的构造函数导致现有的代码不能支持执行了，下面的代码段揭示了这个问题：

```

var maneuver = function(vehicle) {
    write(vehicle.state());
    vehicle.start();
    write(vehicle.state());
    vehicle.accelerate();
    write(vehicle.state());
    write(vehicle.speed());
    vehicle.decelerate();
    write(vehicle.speed());
    if (vehicle.state() != "idle") {
        throw "The vehicle is still moving!";
    }
    vehicle.stop();
    write(vehicle.state());
};

```

根据上面的代码，我们看到抛出的异常是“The vehicle is still moving!”，这是因为写这段代码的作者一直认为加速（accelerate）和减速（decelerate）的数字是一样的。但

FastVehicle的代码和Vehicle的代码并不是完全能够替换掉的。因此，FastVehicle违反了里氏替换原则。

在这点上，你可能会想：“但，客户端不能老假定vehicle都是按照这样的规则来做”，里氏替换原则(LSP)的妨碍（译者注：就是妨碍实现LSP的代码）不是基于我们所想的继承子类应该在行为里确保更新代码，而是这样的更新是否能在当前的期望中得到实现。

上述代码这个case，解决这个不兼容的问题需要在vehicle类库或者客户端调用代码上进行一点重新设计，或者两者都要改。

减少LSP妨碍

那么，我们如何避免LSP妨碍？不幸的话，并不是一直都是可以做到的。我们这里有几个策略我们处理这个事情。

契约 (Contracts)

处理LSP过分妨碍的一个策略是使用契约，契约清单有2种形式：执行说明书（executable specifications）和错误处理，在执行说明书里，一个详细类库的契约也包括一组自动化测试，而错误处理是在代码里直接处理的，例如在前置条件，后置条件，常量检查等，可以从Bertrand Miller的大作《[契约设计](#)》中查看这个技术。虽然自动化测试和契约设计不在本篇文章的范围内，但当我们用的时候我还是推荐如下内容：

1. 检查使用测试驱动开发（Test-Driven Development）来指导你代码的设计
2. 设计可重用类库的时候可随意使用契约设计技术

对于你自己要维护和实现的代码，使用契约设计趋向于添加很多不必要的代码，如果你要控制输入，添加测试是非常有必要的，如果你是类库作者，使用契约设计，你要注意不正确的使用方法以及让你的用户使之作为一个测试工具。

避免继承

避免LSP妨碍的另外一个测试是：如果可能的话，尽量不用继承，在Gamma的大作

《[Design Patterns – Elements of Reusable Object-Oriented Software](#)》中，我们可以看到如下建议：

Favor object composition over class inheritance

尽量使用对象组合而不是类继承

有些书里讨论了组合比继承好的唯一作用是静态类型，基于类的语言（例如，在运行时可以改变行为），与JavaScript相关的一个问题是耦合，当使用继承的时候，继承子类型和他们
本文档使用 [看云](#) 构建

的基类型耦合在一起了，就是说及类型的改变会影响到继承子类型。组合倾向于对象更小化，更容易想静态和动态语言语言维护。

与行为有关，而不是继承

到现在，我们讨论了和继承上下文在内的里氏替换原则，指示出JavaScript的面向对象实。不过，里氏替换原则（LSP）的本质不是真的和继承有关，而是行为兼容性。JavaScript是一个动态语言，一个对象的契约行为不是对象的类型决定的，而是对象期望的功能决定的。里氏替换原则的初始构想是作为继承的一个原则指南，等价于对象设计中的隐式接口。

举例来说，让我们来看一下Robert C. Martin的大作《[敏捷软件开发 原则、模式与实践](#)》中的一个矩形类型：

矩形例子

考虑我们有一个程序用到下面这样的一个矩形对象：

```
var rectangle = {
  length: 0,
  width: 0
};
```

过后，程序有需要一个正方形，由于正方形就是一个长(length)和宽(width)都一样的特殊矩形，所以我们觉得创建一个正方形代替矩形。我们添加了length和width属性来匹配矩形的声明，但我们觉得使用属性的getters/setters一般我们可以让length和width保存同步，确保声明的是一个正方形：

```
var square = {};
(function() {
  var length = 0, width = 0;
  // 注意defineProperty方式是262-5版的新特性
  Object.defineProperty(square, "length", {
    get: function() { return length; },
    set: function(value) { length = width = value; }
  });
  Object.defineProperty(square, "width", {
    get: function() { return width; },
    set: function(value) { length = width = value; }
  });
})();
```

不幸的是，当我们使用正方形代替矩形执行代码的时候发现了问题，其中一个计算矩形面积的方法如下：

```
var g = function(rectangle) {  
    rectangle.length = 3;  
    rectangle.width = 4;  
    write(rectangle.length);  
    write(rectangle.width);  
    write(rectangle.length * rectangle.width);  
};
```

该方法在调用的时候，结果是16，而不是期望的12，我们的正方形square对象违反了LSP原则，square的长度和宽度属性暗示着并不是和矩形100%兼容，但我们并不总是这样明确的暗示。解决这个问题，我们可以重新设计一个shape对象来实现程序，依据多边形的概念，我们声明rectangle和square，relevant。不管怎么说，我们的目的是要说里氏替换原则并不只是继承，而是任何方法（其中的行为可以另外的行为）。

总结

里氏替换原则（LSP）表达的意思不是继承的关系，而是任何方法（只要该方法的行为能体会另外的行为就行）。

(9) 根本没有 “JSON对象” 这回事！

前言

写这篇文章的目的是经常看到开发人员说：把字符串转化为JSON对象，把JSON对象转化成字符串等类似的话题，所以把之前收藏的一篇老外的文章整理翻译了一下，供大家讨论，如有错误，请大家指出，多谢。

正文

本文的主题是基于ECMAScript262-3来写的，2011年的262-5新规范增加了JSON对象，和我们平时所说的JSON有关系，但是不是同一个东西，文章最后一节会讲到新增加的JSON对象。

英文原文：<http://benalman.com/news/2010/03/theres-no-such-thing-as-a-json/>

我想给大家澄清一下一个非常普遍的误解，我认为很多JavaScript开发人员都错误地把JavaScript对象字面量（Object Literals）称为JSON对象（JSON Objects），因为他的语法和JSON规范里描述的一样，但是该规范里也明确地说了JSON只是一个数据交换语言，只有我们将之用在string上下文的时候它才叫JSON。

序列化与反序列化

2个程序（或服务器、语言等）需要交互通信的时候，他们倾向于使用string字符串因为string在很多语言里解析的方式都差不多。复杂的数据结构经常需要用到，并且通过各种各样的中括号{}，小括号()，叫括号<>和空格来组成，这个字符串仅仅是按照要求规范好的字符。

为此，我们为了描述这些复杂的数据结构作为一个string字符串，制定了标准的规则和语法。JSON只是其中一种语法，它可以在string上下文里描述对象，数组，字符串，数字，布尔型和null，然后通过程序间传输，并且反序列化成所需要的格式。YAML和XML（甚至request params）也是流行的数据交换格式，但是，我们喜欢JSON，谁叫我们是JavaScript开发人员呢！

字面量

引用Mozilla Developer Center里的几句话，供大家参考：

1. 他们是固定的值，不是变量，让你从“字面上”理解脚本。（Literals）
2. 字符串字面量是由双引号（"）或单引号（'）包围起来的零个或多个字符组成的。

(Strings Literals)

3. 对象字面量是由大括号 ({}) 括起来的零个或多个对象的属性名-值对。(Object Literals)

何时是JSON，何时不是JSON？

JSON是设计成描述数据交换格式的，他也有自己的语法，这个语法是JavaScript的一个子集。

{ "prop": "val" } 这样的声明有可能是JavaScript对象字面量也有可能是JSON字符串，取决于什么上下文使用它，如果是用在string上下文（用单引号或双引号引住，或者从text文件读取）的话，那它就是JSON字符串，如果是用在对象字面量上下文中，那它就是对象字面量。

```
// 这是JSON字符串
var foo = '{ "prop": "val" }';

// 这是对象字面量
var bar = { "prop": "val" };
```

而且要注意，JSON有非常严格的语法，在string上下文里{ "prop": "val" }是个合法的JSON，但{ prop: "val" }和{ 'prop': 'val' }确实不合法的。所有属性名称和它的值都必须用双引号引住，不能使用单引号。另外，即便你用了转义以后的单引号也是不合法的，详细的语法规则可以到[这里查看](#)。

放到上下文里来看

大家伙可能嗤之以鼻：难道JavaScript代码不是一个大的字符串？

当然是，所有的JavaScript代码和HTML（可能还有其他东西）都是字符串，直到浏览器对他们进行解析。这时候.js文件或者inline的JavaScript代码已经不是字符串了，而是被当成真正的JavaScript源代码了，就像页面里的innerHTML一样，这时候也不是字符串了，而是被解析成DOM结构了。

再次说一下，这取决于上下文，在string上下文里使用带有大括号的JavaScript对象，那它就是JSON字符串，而如果在对象字面量上下文里使用的话，那它就是对象字面量。

真正的JSON对象

开头已经提到，对象字面量不是JSON对象，但是有[真正的JSON对象](#)。但是两者完全不一样概念，在新版的浏览器里JSON对象已经被原生的内置对象了，目前有2个静态方法：JSON.parse用来将JSON字符串反序列化成对象，JSON.stringify用来将对象序列化成JSON字符串。老版本的浏览器不支持这个对象，但你可以通过[json2.js](#)来实现同样的功能。

如果还不理解，别担心，参考一下的例子就知道了：

```
// 这是JSON字符串，比如从AJAX获取字符串信息
var my_json_string = '{ "prop": "val" }';

// 将字符串反序列化成对象
var my_obj = JSON.parse( my_json_string );

alert( my_obj.prop == 'val' ); // 提示 true，和想象的一样！

// 将对象序列化成JSON字符串
var my_other_json_string = JSON.stringify( my_obj );
```

另外，[Paul Irish](#)提到Douglas Crockford在[JSON RFC](#)里用到了 “JSON object”，但是在那个上下文里，他的意思是 “对象描述成JSON字符串” 不是 “对象字面量”。

更多资料

如果你想了解更多关于JSON的资料，下面的连接对你绝对有用：

- [JSON specification](#)
- [JSON RFC](#)
- [JSON on Wikipedia](#)
- [JSONLint - The JSON Validator](#)
- [JSON is not the same as JSON](#)

(10) JavaScript核心 (晋级高手必读篇)

本篇是[ECMA-262-3 in detail](#)系列的一个概述 (本人后续会翻译整理这些文章到本系列 (第11-19章))。每个章节都有一个更详细的内容链接, 你可以继续读一下每个章节对应的详细内容链接进行更深入的了解。

适合的读者: 有经验的开发员, 专业前端人员。

原作者: Dmitry A. Soshnikov

发布时间: 2010-09-02

原文: <http://dmitrysoshnikov.com/ecmascript/javascript-the-core/>

参考1: <http://ued.ctrip.com/blog/?p=2795>

参考2: <http://www.cnblogs.com/ifishing/archive/2010/12/08/1900594.html>

主要是综合了上面2位高手的中文翻译, 将两篇文章的精华部分都结合在一起了。

我们首先来看一下对象[Object]的概念, 这也是ECMAScript中最基本的概念。

对象Object

ECMAScript是一门高度抽象的面向对象(object-oriented)语言, 用以处理Objects对象. 当然, 也有基本类型, 但是必要时, 也需要转换成object对象来用。

An object is a collection of properties and has a single prototype object. The prototype may be either an object or the null value.

Object是一个属性的集合, 并且都拥有一个单独的原型对象[prototype object]. 这个原型对象[prototype object]可以是一个object或者null值。

让我们来举一个基本Object的例子, 首先我们要清楚, 一个Object的prototype是一个内部的[[prototype]]属性的引用。

不过一般来说, 我们会使用__下划线来代替双括号, 例如proto__(这是某些脚本引擎比如SpiderMonkey的对于原型概念的具体实现, 尽管并非标准)。

```
var foo = {  
  x: 10,
```

本文档使用 [看云](#) 构建

```

    y: 20
  };

```

上述代码foo对象有两个显式的属性[explicit own properties]和一个自带隐式的 proto 属性 [implicit __proto__ property]，指向foo的原型。

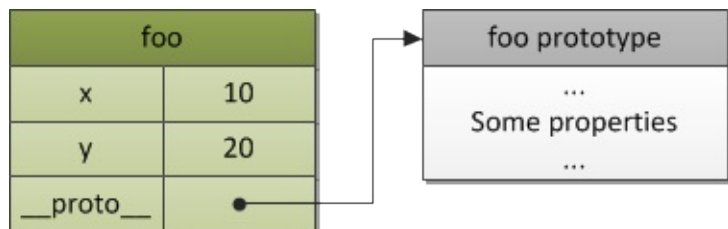


图 1. 一个含有原型的基本对象

为什么需要原型呢，让我们考虑 原型链 的概念来回答这个问题。

原型链 (Prototype chain)

原型对象也是普通的对象，并且也有可能有自己的原型，如果一个原型对象的原型不为null的话，我们就称之为原型链 (prototype chain)。

A prototype chain is a finite chain of objects which is used to implemented inheritance and shared properties.

原型链是一个由对象组成的有限对象链由于实现继承和共享属性。

想象一个这种情况，2个对象，大部分内容都一样，只有一小部分不一样，很明显，在一个好的设计模式中，我们会需要重用那部分相同的，而不是在每个对象中重复定义那些相同的方法或者属性。在基于类[class-based]的系统中，这些重用部分被称为类的继承 – 相同的部分放入class A，然后class B和class C从A继承，并且可以声明拥有各自的独特的东西。

ECMAScript没有类的概念。但是，重用[reuse]这个理念没什么不同（某些方面，甚至比class-更加灵活），可以由prototype chain原型链来实现。这种继承被称为delegation based inheritance-基于继承的委托，或者更通俗一些，叫做原型继承。

类似于类“ A ”，“ B ”，“ C ”，在ECMAScript中尼创建对象类“ a ”，“ b ”，“ c ”，相应地，对象“a”拥有对象“b”和“c”的共同部分。同时对象“b”和“c”只包含它们自己的附加属性或方法。

```

var a = {
  x: 10,

```

本文档使用 [看云](#) 构建

```

    calculate: function (z) {
        return this.x + this.y + z
    }
};

var b = {
    y: 20,
    __proto__: a
};

var c = {
    y: 30,
    __proto__: a
};

// 调用继承过来的方法
b.calculate(30); // 60
c.calculate(40); // 80

```

这样看上去是不是很简单啦。b和c可以使用a中定义的calculate方法，这就是有原型链来[prototype chain]实现的。

原理很简单:如果在对象b中找不到calculate方法(也就是对象b中没有这个calculate属性), 那么就会沿着原型链开始找。如果这个calculate方法在b的prototype中没有找到, 那么就会沿着原型链找到a的prototype, 一直遍历完整个原型链。记住, 一旦找到, 就返回第一个找到的属性或者方法。因此, 第一个找到的属性成为继承属性。如果遍历完整个原型链, 仍然没有找到, 那么就会返回undefined。

注意一点, this这个值在一个继承机制中, 仍然是指向它原本属于的对象, 而不是从原型链上找到它时, 它所属于的对象。例如, 以上的例子, this.y是从b和c中获取的, 而不是a。当然, 你也发现了this.x是从a取的, 因为是通过原型链机制找到的。

如果一个对象的prototype没有显示的声明过或定义过, 那么prototype的默认值就是object.prototype, 而object.prototype也会有一个prototype, 这个就是原型链的终点了, 被设置为null。

下面的图示就是表示了上述a,b,c的继承关系

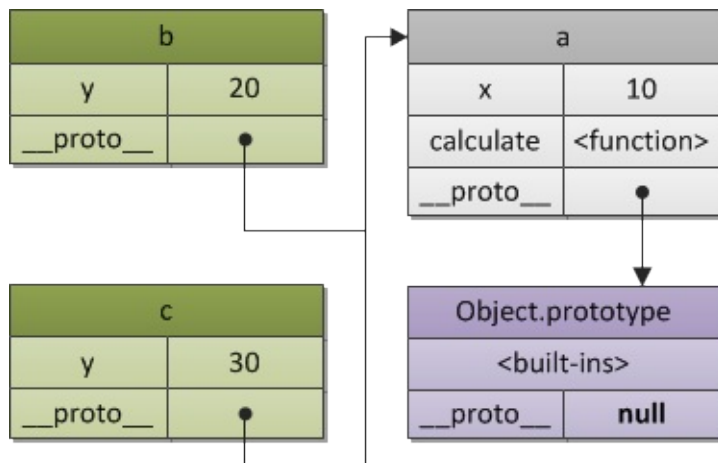


图 2. 原型链

原型链通常将会在这样的情况下使用：对象拥有 相同或相似的状态结构(same or similar state structure)（即相同的属性集合）与 不同的状态值(different state values)。在这种情况下，我们可以使用 构造函数(Constructor) 在 特定模式(specified pattern) 下创建对象。

构造函数(Constructor)

除了创建对象，构造函数(constructor) 还做了另一件有用的事情—自动为创建的新对象设置了原型对象(prototype object)。原型对象存放于 ConstructorFunction.prototype 属性中。

例如，我们重写之前例子，使用构造函数创建对象 “b” 和 “c”，那么对象 “a” 则扮演了 “Foo.prototype” 这个角色：

```

// 构造函数
function Foo(y) {
  // 构造函数将会以特定模式创建对象：被创建的对象都会有"y"属性
  this.y = y;
}

// "Foo.prototype"存放了新建对象的原型引用
// 所以我们可以将之用于定义继承和共享属性或方法
// 所以，和上例一样，我们有了如下代码：

// 继承属性"x"
Foo.prototype.x = 10;

// 继承方法"calculate"
Foo.prototype.calculate = function (z) {
  return this.x + this.y + z;
};

// 使用foo模式创建 "b" and "c"

```

```

var b = new Foo(20);
var c = new Foo(30);

// 调用继承的方法
b.calculate(30); // 60
c.calculate(40); // 80

// 让我们看看是否使用了预期的属性

console.log(

  b.__proto__ === Foo.prototype, // true
  c.__proto__ === Foo.prototype, // true

  // "Foo.prototype"自动创建了一个特殊的属性"constructor"
  // 指向a的构造函数本身
  // 实例"b"和"c"可以通过授权找到它并用以检测自己的构造函数

  b.constructor === Foo, // true
  c.constructor === Foo, // true
  Foo.prototype.constructor === Foo // true

  b.calculate === b.__proto__.calculate, // true
  b.__proto__.calculate === Foo.prototype.calculate // true

);

```

上述代码可表示为如下的关系：

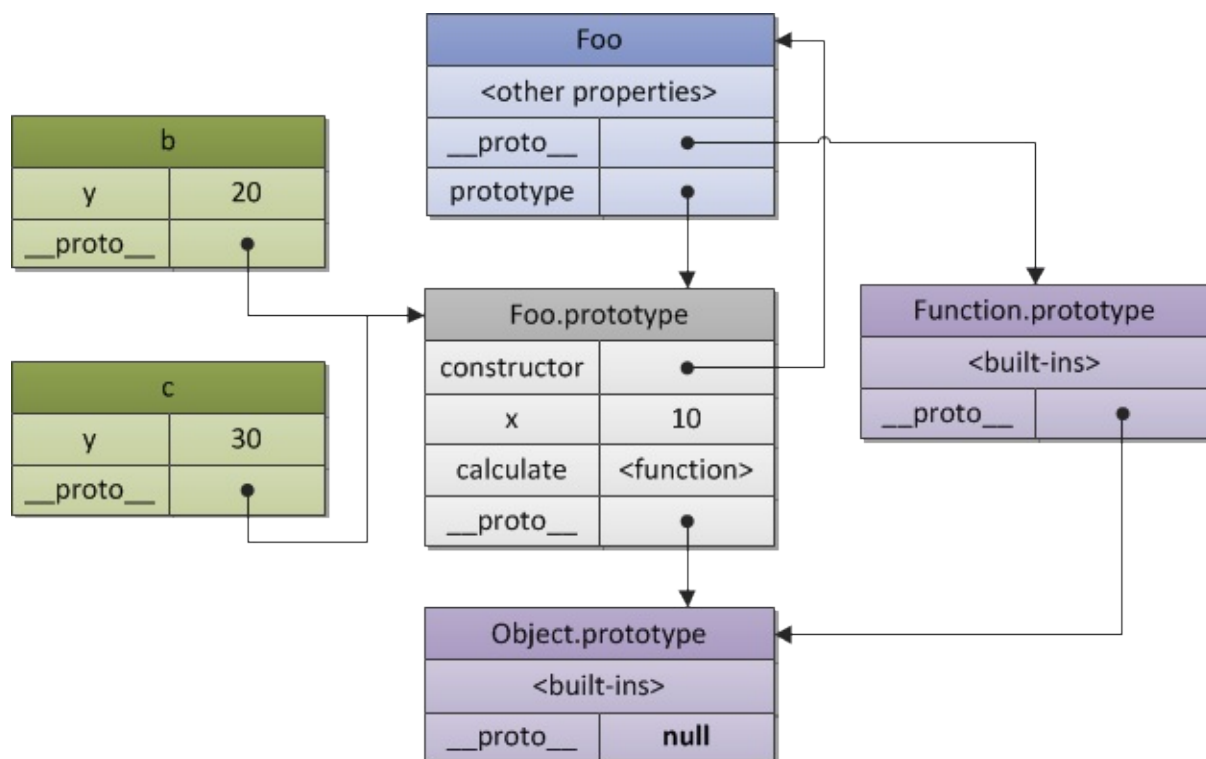


图 3. 构造函数与对象之间的关系

上述图示可以看出，每一个object都有一个prototype. 构造函数Foo也拥有自己的proto, 也就是Function.prototype, 而Function.prototype的proto指向了Object.prototype. 重申一遍，Foo.prototype只是一个显式的属性，也就是b和c的proto属性。

这个问题完整和详细的解释可以在大叔即将翻译的第18、19两章找到。有两个部分：面向对象编程.一般理论(OOP. The general theory)，描述了不同的面向对象的范式与风格(OOP paradigms and stylistics)，以及与ECMAScript的比较, 面向对象编程.ECMAScript实现(OOP. ECMAScript implementation), 专门讲述了ECMAScript中的面向对象编程。

现在，我们已经了解了基本的object原理，那么我们接下去来看看ECMAScript里面的程序执行环境[runtime program execution]. 这就是通常称为的“执行上下文堆栈” [execution context stack]。每一个元素都可以抽象的理解为object。你也许发现了，没错，在ECMAScript中，几乎处处都能看到object的身影。

执行上下文栈(Execution Context Stack)

在ECMAScript中的代码有三种类型：global, function和eval。

每一种代码的执行都需要依赖自身的上下文。当然global的上下文可能涵盖了很多的function和eval的实例。函数的每一次调用，都会进入函数执行中的上下文,并且来计算函数中变量等的值。eval函数的每一次执行，也会进入eval执行中的上下文，判断应该从何处获取变量的值。

注意，一个function可能产生无限的上下文环境，因为一个函数的调用（甚至递归）都产生了一个新的上下文环境。

```
function foo(bar) {}

// 调用相同的function，每次都会产生3个不同的上下文
// （包含不同的状态，例如参数bar的值）

foo(10);
foo(20);
foo(30);
```

一个执行上下文可以激活另一个上下文，就好比一个函数调用了另一个函数(或者全局的上下文调用了一个全局函数)，然后一层一层调用下去。逻辑上来说，这种实现方式是栈，我们可以称之为上下文堆栈。

激活其它上下文的某个上下文被称为 调用者(caller)。被激活的上下文被称为被调用者(callee)。被调用者同时也可能是调用者(比如一个在全局上下文中被调用的函数调用某些自身的内部方法)。

当一个caller激活了一个callee，那么这个caller就会暂停它自身的执行，然后将控制权交给这个callee。于是这个callee被放入堆栈，称为进行中的上下文[running/active execution context]。当这个callee的上下文结束之后，会把控制权再次交给它的caller，然后caller会在刚才暂停的地方继续执行。在这个caller结束之后，会继续触发其他的上下文。一个callee可以用返回 (return) 或者抛出异常 (exception) 来结束自身的上下文。

如下图，所有的ECMAScript的程序执行都可以看做是一个执行上下文堆栈[execution context (EC) stack]。堆栈的顶部就是处于激活状态的上下文。

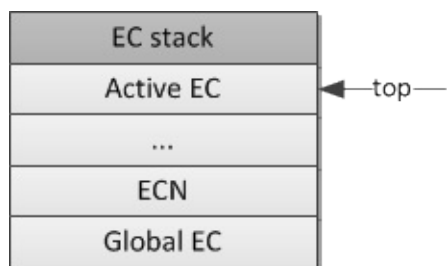


图 4. 执行上下文栈

当一段程序开始时，会先进入全局执行上下文环境[global execution context]，这个也是堆栈中最底部的元素。此全局程序会开始初始化，初始化生成必要的对象[objects]和函数[functions]。在此全局上下文执行的过程中，它可能会激活一些方法（当然是已经初始化过的），然后进入他们的上下文环境，然后将新的元素压入堆栈。在这些初始化都结束之后，这个系统会等待一些事件（例如用户的鼠标点击等），会触发一些方法，然后进入一个新的上下文环境。

见图5，有一个函数上下文“EC1”和一个全局上下文“Global EC”，下图展现了从“Global EC”进入和退出“EC1”时栈的变化：

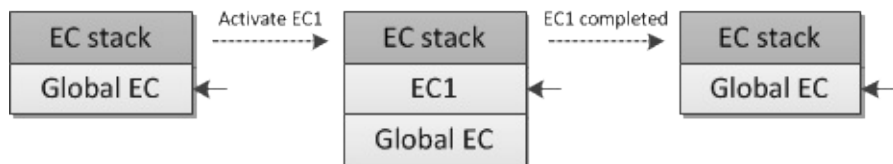


图 5. 执行上下文栈的变化

ECMAScript运行时系统就是这样管理代码的执行。

关于ECMAScript执行上下文栈的内容请查阅本系列教程的第11章执行上下文(Execution context)。

如上所述，栈中每一个执行上下文可以表示为一个对象。让我们看看上下文对象的结构以及执行其代码所需的 状态(state) 。

执行上下文(Execution Context)

一个执行的上下文可以抽象的理解为object。每一个执行的上下文都有一系列的属性（我们称为上下文状态），他们用来追踪关联代码的执行进度。这个图示就是一个context的结构。

Execution context	
Variable object	{ vars, function declarations, arguments... }
Scope chain	[Variable object + all parent scopes]
thisValue	Context object

图 6. 上下文结构

除了这3个所需要的属性(变量对象(variable object)，this指针(this value)，作用域链(scope chain))，执行上下文根据具体实现还可以具有任意额外属性。接着，让我们仔细来看看这三个属性。

变量对象(Variable Object)

A variable object is a scope of data related with the execution context.

It' s a special object associated with the context and which stores variables and function declarations are being defined within the context.

变量对象(variable object) 是与执行上下文相关的 数据作用域(scope of data) 。

它是与上下文关联的特殊对象，用于存储被定义在上下文中的 变量(variables) 和 函数声明(function declarations) 。

注意：函数表达式[function expression] (而不是函数声明[function declarations , 区别请参考[本系列第2章]

(<http://www.cnblogs.com/TomXu/archive/2011/12/29/2290308.html>)) 是不包含在VO[variable object]里面的。

变量对象 (Variable Object) 是一个抽象的概念，不同的上下文中，它表示使用不同的object。例如，在global全局上下文中，变量对象也是全局对象自身[global object]。（这就是我们可以通过全局对象的属性来指向全局变量）。

让我们看看下面例子中的全局执行上下文情况：

```
var foo = 10;

function bar() {} // // 函数声明
(function baz() {}); // 函数表达式

console.log(
  this.foo == foo, // true
  window.bar == bar // true
);

console.log(baz); // 引用错误, baz没有被定义
```

全局上下文中的变量对象(VO)会有如下属性：

Global VO	
foo	10
bar	<function>
<built-ins>	

图 7. 全局变量对象

如上所示，函数“baz”如果作为函数表达式则不被不被包含于变量对象。这就是在函数外部尝试访问产生引用错误(ReferenceError)的原因。请注意，ECMAScript和其他语言相比(比如C/C++)，仅有函数能够创建新的作用域。在函数内部定义的变量与内部函数，在外部非直接可见并且不污染全局对象。使用eval的时候，我们同样会使用一个新的(eval创建)执行上下文。eval会使用全局变量对象或调用者的变量对象(eval的调用来源)。

那函数以及自身的变量对象又是怎样的呢?在一个函数上下文中，变量对象被表示为活动对象

(activation object)。

活动对象(activation object)

当函数被调用者激活，这个特殊的活动对象(activation object) 就被创建了。它包含普通参数(formal parameters) 与特殊参数(arguments)对象(具有索引属性的参数映射表)。活动对象在函数上下文中作为变量对象使用。

即：函数的变量对象保持不变，但除去存储变量与函数声明之外，还包含以及特殊对象arguments。

考虑下面的情况：

```
function foo(x, y) {  
  var z = 30;  
  function bar() {} // 函数声明  
  (function baz() {}); // 函数表达式  
}  
  
foo(10, 20);
```

“foo” 函数上下文的下一个激活对象(AO)如下图所示：

Activation object	
x	10
y	20
arguments	{0: 10, 1: 20, ..}
z	30
bar	<function>

图 8. 激活对象

同样道理，function expression不在AO的行列。

对于这个AO的详细内容可以通过本系列教程第9章找到。

我们接下去要讲到的是第三个主要对象。众所周知，在ECMAScript中，我们会用到内部函数[inner functions]，在这些内部函数中，我们可能会引用它的父函数变量，或者全局的变量。我们把这些变量对象成为上下文作用域对象[scope object of the context]. 类似于上面

讨论的原型链[prototype chain]，我们在这里称为作用域链[scope chain]。

作用域链(Scope Chains)

A scope chain is a list of objects that are searched for identifiers appear in the code of the context.

作用域链是一个 对象列表(list of objects) ，用以检索上下文代码中出现的 标识符(identifiers) 。

作用域链的原理和原型链很类似，如果这个变量在自己的作用域中没有，那么它会寻找父级的，直到最顶层。

标示符[Identifiers]可以理解为变量名称、函数声明和普通参数。例如，当一个函数在自身函数体内需要引用一个变量，但是这个变量并没有在函数内部声明（或者也不是某个参数名），那么这个变量就可以称为自由变量[free variable]。那么我们搜寻这些自由变量就需要用到作用域链。

在一般情况下，一个作用域链包括父级变量对象（variable object）（作用域链的顶部）、函数自身变量VO和活动对象（activation object）。不过，有些情况下也会包含其它的对象，例如在执行期间，动态加入作用域链中的一例如with或者catch语句。[译注：with-objects指的是with语句，产生的临时作用域对象；catch-clauses指的是catch从句，如catch(e)，这会产生异常对象，导致作用域变更]。

当查找标识符的时候，会从作用域链的活动对象部分开始查找，然后(如果标识符没有在活动对象中找到)查找作用域链的顶部，循环往复，就像作用域链那样。

```
var x = 10;

(function foo() {
  var y = 20;
  (function bar() {
    var z = 30;
    // "x"和"y"是自由变量
    // 会在作用域链的下一个对象中找到（函数"bar"的互动对象之后）
    console.log(x + y + z);
  })();
})();
```

我们假设作用域链的对象联动是通过一个叫做parent的属性，它是指向作用域链的下一个对象。这可以在Rhino Code中测试一下这种流程，这种技术也确实在ES5环境中实现了(有一

个称为outer链接).当然也可以用一个简单的数据来模拟这个模型。使用parent的概念,我们可以把上面的代码演示成如下的情况。(因此, 父级变量是被存在函数的[[Scope]]属性中的)。

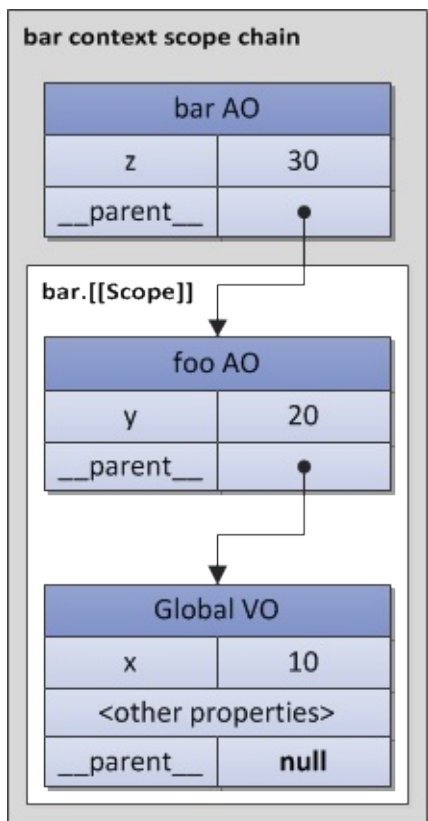


图 9. 作用域链

在代码执行过程中,如果使用with或者catch语句就会改变作用域链。而这些对象都是一些简单对象,他们也会有原型链。这样的话,作用域链会从两个维度来搜寻。

1. 首先在原本的作用域链
2. 每一个链接点的作用域的链 (如果这个链接点是有prototype的话)

我们再看下面这个例子：

```
Object.prototype.x = 10;

var w = 20;
var y = 30;

// 在SpiderMonkey全局对象里
// 例如, 全局上下文的变量对象是从"Object.prototype"继承到的
// 所以我们可以得到“没有声明的全局变量”
```

```
// 因为可以从原型链中获取

console.log(x); // 10

(function foo() {

    // "foo" 是局部变量
    var w = 40;
    var x = 100;

    // "x" 可以从"Object.prototype"得到, 注意值是10哦
    // 因为{z: 50}是从它那里继承的

    with ({z: 50}) {
        console.log(w, x, y , z); // 40, 10, 30, 50
    }

    // 在"with"对象从作用域链删除之后
    // x又可以从foo的上下文中得到了, 注意这次值又回到了100哦
    // "w" 也是局部变量
    console.log(x, w); // 100, 40

    // 在浏览器里
    // 我们可以通过如下语句来得到全局的w值
    console.log(window.w); // 20

})();
```

我们就会有如下结构图示。这表示, 在我们去搜寻parent之前, 首先会去proto的链接中。

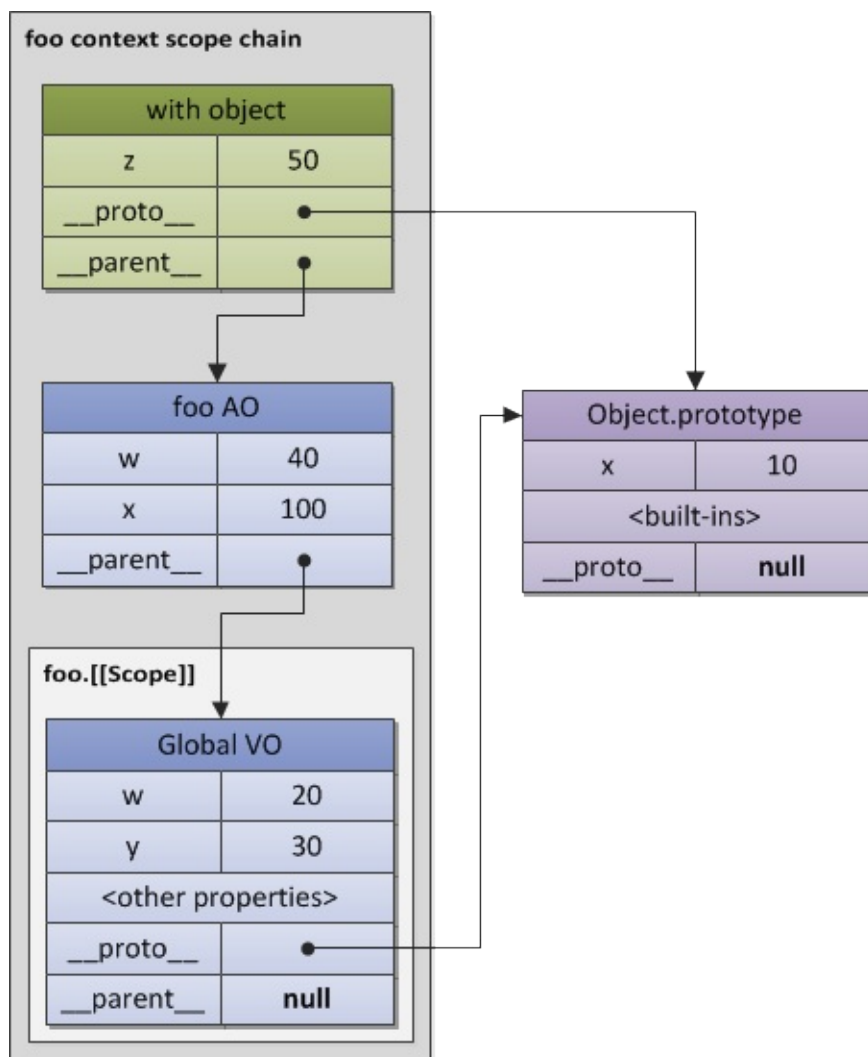


图 10. with增大的作用域链

注意，不是所有的全局对象都是由`Object.prototype`继承而来的。上述图示的情况可以在SpiderMonkey中测试。

只要所有外部函数的变量对象都存在，那么从内部函数引用外部数据则没有特别之处——我们只要遍历作用域链表，查找所需变量。然而，如上文所提及，当一个上下文终止之后，其状态与自身将会被销毁(destroyed)，同时内部函数将会从外部函数中返回。此外，这个返回的函数之后可能会在其他的上下文中被激活，那么如果一个之前被终止的含有一些自由变量的上下文又被激活将会怎样？通常来说，解决这个问题的概念在ECMAScript中与作用域链直接相关，被称为(词法)闭包((lexical) closure)。

闭包(Closures)

在ECMAScript中，函数是“第一类”对象。这个名词意味着函数可以作为参数被传递给其

他函数使用 (在这种情况下, 函数被称为 “funargs” —— “functional arguments” 的缩写 [译注: 这里不知翻译为泛函参数是否恰当])。接收 “funargs” 的函数被称之为 高阶函数 (higher-order functions), 或者更接近数学概念的话, 被称为 运算符(operators)。其他函数的运行时也会返回函数, 这些返回的函数被称为 function valued 函数 (有 functional value 的函数)。

“funargs” 与 “functional values” 有两个概念上的问题, 这两个子问题被称为 “Funarg problem” (“泛函参数问题”)。要准确解决泛函参数问题, 需要引入 闭包(closures) 到的概念。让我们仔细描述这两个问题(我们可以见到, 在ECMAScript中使用了函数的[[Scope]] 属性来解决这个问题)。

“funarg problem” 的一个子问题是 “upward funarg problem” [译注: 或许可以翻译为: 向上查找的函数参数问题]。当一个函数从其他函数返回到外部的时候, 这个问题将会出现。要能够在外部上下文结束时, 进入外部上下文的变量, 内部函数 在创建的时候(at creation moment) 需要将之存储进[[Scope]]属性的父元素的作用域中。然后当函数被激活时, 上下文的作用域链表现为激活对象与[[Scope]]属性的组合(事实上, 可以在上图见到):

Scope chain = Activation object + [[Scope]]

作用域链 = 活动对象 + [[Scope]]

请注意, 最主要的事情是——函数在被创建时保存外部作用域, 是因为这个 被保存的作用域链(saved scope chain) 将会在未来的函数调用中用于变量查找。

```
function foo() {
  var x = 10;
  return function bar() {
    console.log(x);
  };
}

// "foo"返回的也是一个function
// 并且这个返回的function可以随意使用内部的变量x

var returnedFunction = foo();

// 全局变量 "x"
var x = 20;

// 支持返回的function
returnedFunction(); // 结果是10而不是20
```

这种形式的作用域称为静态作用域[static/lexical scope]。上面的x变量就是在函数bar的[[Scope]]中搜寻到的。理论上来说，也会有动态作用域[dynamic scope]，也就是上述的x被解释为20，而不是10。但是ECMAScript不使用动态作用域。

“funarg problem” 的另一个类型就是自上而下[“ downward funarg problem”]。在这种情况下，父级的上下会存在，但是在判断一个变量值的时候会存在多义性。也就是，这个变量究竟应该使用哪个作用域。是在函数创建时的作用域呢，还是在执行时的作用域呢？为了避免这种多义性，可以采用闭包，也就是使用静态作用域。

请看下面的例子：

```
// 全局变量 "x"
var x = 10;

// 全局function
function foo() {
  console.log(x);
}

(function (funArg) {

  // 局部变量 "x"
  var x = 20;

  // 这不会有歧义
  // 因为我们使用"foo"函数的[[Scope]]里保存的全局变量"x",
  // 并不是caller作用域的"x"

  funArg(); // 10, 而不是20

})(foo); // 将foo作为一个"funarg"传递下去
```

从上述的情况，我们似乎可以断定，在语言中，使用静态作用域是闭包的一个强制性要求。不过，在某些语言中，会提供动态和静态作用域的结合，可以允许开发人员选择哪一种作用域。但是在ECMAScript中，只采用了静态作用域。所以ECMAScript完全支持使用[[Scope]]的属性。我们可以给闭包得出如下定义：

A closure is a combination of a code block (in ECMAScript this is a function) and statically/lexically saved all parent scopes.

Thus, via these saved scopes a function may easily refer free variables.

闭包是一系列代码块（在ECMAScript中是函数），并且静态保存所有父级的作用域。通过这些保存的作用域来搜寻到函数中的自由变量。

请注意，因为每一个普通函数在创建时保存了[[Scope]]，理论上，ECMAScript中所有函数都是闭包。

还有一个很重要的点，几个函数可能含有相同的父级作用域（这是一个很普遍的情况，例如有好几个内部或者全局的函数）。在这种情况下，在[[Scope]]中存在的变量是会共享的。一个闭包中变量的变化，也会影响另一个闭包的。

```
function baz() {
  var x = 1;
  return {
    foo: function foo() { return ++x; },
    bar: function bar() { return --x; }
  };
};

var closures = baz();

console.log(
  closures.foo(), // 2
  closures.bar() // 1
);
```

上述代码可以用这张图来表示：

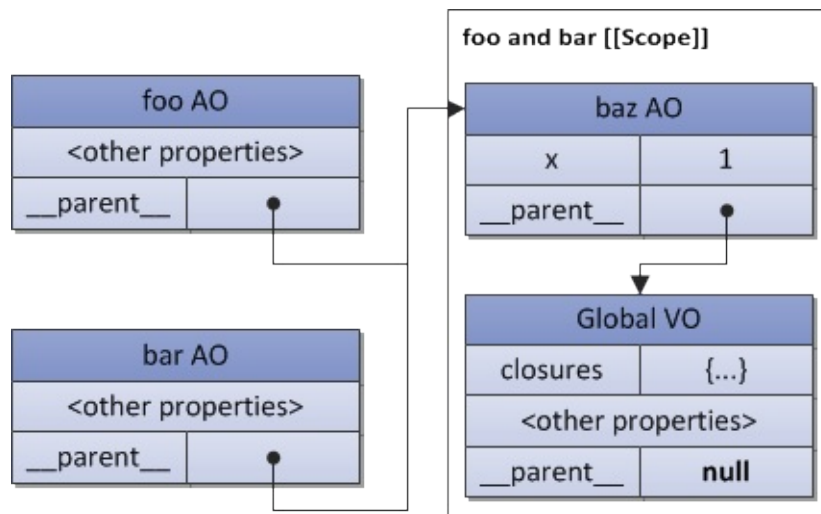


图 11. 共享的[[Scope]]

在某个循环中创建多个函数时，上图会引发一个困惑。如果在创建的函数中使用循环变量（如“k”），那么所有的函数都使用同样的循环变量，导致一些程序员经常会得不到预期值。现在清楚为什么会产生如此问题了——因为所有函数共享同一个[[Scope]]，其中循环变量为最后一次复赋值。

```
var data = [];

for (var k = 0; k < 3; k++) {
  data[k] = function () {
    alert(k);
  };
}

data[0](); // 3, but not 0
data[1](); // 3, but not 1
data[2](); // 3, but not 2
```

有一些用以解决这类问题的技术。其中一种技巧是在作用域链中提供一个额外的对象，比如增加一个函数：

```
var data = [];

for (var k = 0; k < 3; k++) {
  data[k] = (function (x) {
    return function () {
      alert(x);
    };
  })(k); // 将k当做参数传递进去
}

// 结果正确
data[0](); // 0
data[1](); // 1
data[2](); // 2
```

闭包理论的深入研究与具体实践可以在本系列教程第16章闭包(Closures)中找到。如果想得到关于作用域链的更多信息，可以参照本系列教程第14章作用域链(Scope chain)。

下一章节将会讨论一个执行上下文的最后一个属性——this指针的概念。

This指针

A this value is a special object which is related with the execution context. Therefore, it may be named as a context object (i.e. an object in which context the execution context is activated).

this适合执行的上下文环境息息相关的一个特殊对象。因此，它也可以称为上下文对象context object。

任何对象都可以作为上下文的this值。我想再次澄清对与ECMAScript中，与执行上下文相关

的一些描述——特别是this的误解。通常，this 被错误地，描述为变量对象的属性。最近比如在这本书中就发现了(尽管书中提及this的那一章还不错)。请牢记：

a this value is a property of the execution context, but not a property of the variable object.

this是执行上下文环境的一个属性，而不是某个变量对象的属性

这个特点很重要，因为和变量不同，this是没有任何一个类似搜寻变量的过程。当你在代码中使用了this,这个 this的值就直接从执行的上下文中获取了，而不会从作用域链中搜寻。this的值只取决中进入上下文时的情况。

顺便说一句，和ECMAScript不同，Python有一个self的参数，和this的情况差不多，但是可以在执行过程中被改变。在ECMAScript中，是不可以给this赋值的，因为，还是那句话，this不是变量。

在global context(全局上下文)中，this的值就是指全局这个对象，这就意味着，this值就是这个变量本身。

```
var x = 10;

console.log(
  x, // 10
  this.x, // 10
  window.x // 10
);
```

在函数上下文[function context]中，this可能会根据每次的函数调用而成为不同的值.this会由每一次caller提供,caller是通过调用表达式[call expression]产生的（也就是这个函数如何被激活调用的）。例如，下面的例子中foo就是一个callee，在全局上下文中被激活。下面的例子就表明了不同的caller引起this的不同。

```
// "foo"函数里的alert没有改变
// 但每次激活调用的时候this是不同的

function foo() {
  alert(this);
}

// caller 激活 "foo"这个callee,
// 并且提供"this"给这个 callee

foo(); // 全局对象
```

```
foo.prototype.constructor(); // foo.prototype

var bar = {
  baz: foo
};

bar.baz(); // bar

(bar.baz)(); // also bar
(bar.baz = bar.baz)(); // 这是一个全局对象
(bar.baz, bar.baz)(); // 也是全局对象
(false || bar.baz)(); // 也是全局对象

var otherFoo = bar.baz;
otherFoo(); // 还是全局对象
```

如果要深入思考每一次函数调用中，this值的变化(更重要的是怎样变化)，你可以阅读本系列教程第10章This。上文所提及的情况都会在此章内详细讨论。

总结(Conclusion)

在此我们完成了一个简短的概述。尽管看来不是那么简短，但是这些话题若要完整表述完毕，则需要一整本书。我们没有提及两个重要话题：函数(functions) (以及不同类型的函数之间的不同，比如函数声明与函数表达式)与ECMAScript的 求值策略(evaluation strategy)。这两个话题可以分别查阅本系列教程第15章函数(Functions) 与第19章求值策略(Evaluation strategy)。

如果你有任何评论，问题或者补充，我很欢迎在文章评论中讨论。

祝大家学习ECMAScript顺利。

(11) 执行上下文 (Execution Contexts)

简介

从本章开始，我将陆续（翻译、转载、整理）<http://dmitrysoshnikov.com/网站关于ECMAScript标准理解的好文>。

本章我们要讲解的是ECMAScript标准里的执行上下文和相关可执行代码的各种类型。

原始作者：Dmitry A. Soshnikov

原始发布：2009-06-26

俄文原文：<http://dmitrysoshnikov.com/ecmascript/ru-chapter-1-execution-contexts/>

英文翻译：Dmitry A. Soshnikov.

发布时间：2010-03-11

英文翻译：<http://dmitrysoshnikov.com/ecmascript/chapter-1-execution-contexts/>

本文参考了博客园[justinw的中文翻译](#)，做了一些错误修正，感谢译者。

定义

每次当控制器转到ECMAScript可执行代码的时候，即会进入到一个执行上下文。执行上下文(简称-EC)是ECMA-262标准里的一个抽象概念，用于同可执行代码(executable code)概念进行区分。

标准规范没有从技术实现的角度定义EC的准确类型和结构，这应该是具体实现ECMAScript引擎时要考虑的问题。

活动的执行上下文组在逻辑上组成一个堆栈。堆栈底部永远都是全局上下文(global context)，而顶部就是当前(活动的)执行上下文。堆栈在EC类型进入和退出上下文的时候被修改（推入或弹出）。

可执行代码类型

可执行代码的类型这个概念与执行上下文的抽象概念是有关系的。在某些时刻，可执行代码与执行上下文完全有可能是等价的。

例如，我们可以定义执行上下文堆栈是一个数组：

```
ECStack = [];
```

每次进入function (即使function被递归调用或作为构造函数) 的时候或者内置的eval函数工作的时候，这个堆栈都会被压入。

全局代码

这种类型的代码是在"程序"级处理的：例如加载外部的js文件或者本地标签内的代码。全局代码不包括任何function体内的代码。

在初始化（程序启动）阶段，ECStack是这样的：

```
ECStack = [
  globalContext
];
```

函数代码

当进入function函数代码(所有类型的functions)的时候，ECStack被压入新元素。需要注意的是，具体的函数代码不包括内部函数(inner functions)代码。如下所示，我们使函数自己调自己的方式递归一次：

```
(function foo(bar) {
  if (bar) {
    return;
  }
  foo(true);
})();
```

那么，ECStack以如下方式被改变：

```
// 第一次foo的激活调用
ECStack = [
  functionContext
  globalContext
];

// foo的递归激活调用
ECStack = [
  functionContext - recursively
  functionContext
  globalContext
];
```

每次return的时候，都会退出当前执行上下文的，相应地ECStack就会弹出，栈指针会自动移动位置，这是一个典型的堆栈实现方式。一个抛出的异常如果没被截获的话也有可能从一个或多个执行上下文退出。相关代码执行完以后，ECStack只会包含全局上下文(global context)，一直到整个应用程序结束。

Eval 代码

eval 代码有点儿意思。它有一个概念：调用上下文(calling context),例如，eval函数调用的时候产生的上下文。eval(变量或函数声明)活动会影响调用上下文(calling context)。

```
eval('var x = 10');

(function foo() {
  eval('var y = 20');
})();

alert(x); // 10
alert(y); // "y" 提示没有声明
```

ECStack的变化过程：

```
ECStack = [
  globalContext
];

// eval('var x = 10');
ECStack.push(
  evalContext,
  callingContext: globalContext
);

// eval exited context
ECStack.pop();

// foo function call
ECStack.push( functionContext);

// eval('var y = 20');
ECStack.push(
  evalContext,
  callingContext: functionContext
);

// return from eval
ECStack.pop();

// return from foo
ECStack.pop();
```

也就是一个非常普通的逻辑调用堆栈。

在版本号1.7以上的SpiderMonkey(内置于Firefox,Thunderbird)的实现中，可以把调用上下文作为第二个参数传递给eval。那么，如果这个上下文存在，就有可能影响“私有”(有人喜欢这样叫它)变量。

```
function foo() {  
    var x = 1;  
    return function () { alert(x); };  
};  
  
var bar = foo();  
  
bar(); // 1  
  
eval('x = 2', bar); // 传入上下文，影响了内部的var x 变量  
  
bar(); // 2
```

结论

这篇文章是后面分析其他跟执行上下文相关的主题(例如变量对象，作用域链，等等)的最起码的理论基础，这些主题将在后续章节中讲到。

其他参考

这篇文章的内容在ECMA-262-3 标准规范中对应的章节— [10. Execution Contexts](#).

(12) 变量对象 (Variable Object)

介绍

JavaScript编程的时候总避免不了声明函数和变量，以成功构建我们的系统，但是解释器是如何并且在什么地方去查找这些函数和变量呢？我们引用这些对象的时候究竟发生了什么？

原始发布：Dmitry A. Soshnikov

发布时间：2009-06-27

俄文地址：<http://dmitrysoshnikov.com/ecmascript/ru-chapter-2-variable-object/>

英文翻译：Dmitry A. Soshnikov

发布时间：2010-03-15

英文地址：<http://dmitrysoshnikov.com/ecmascript/chapter-2-variable-object/>

部分难以翻译的句子参考了[justinw的中文翻译](#)

大多数ECMAScript程序员应该都知道变量与执行上下文有密切关系：

```
var a = 10; // 全局上下文中的变量

(function () {
  var b = 20; // function上下文中的局部变量
})();

alert(a); // 10
alert(b); // 全局变量 "b" 没有声明
```

并且，很多程序员也都知道，当前ECMAScript规范指出独立作用域只能通过“函数(function)”代码类型的执行上下文创建。也就是说，相对于C/C++来说，ECMAScript里的for循环并不能创建一个局部的上下文。

```
for (var k in {a: 1, b: 2}) {
  alert(k);
}

alert(k); // 尽管循环已经结束但变量k依然在当前作用域
```

我们来看看一下，我们声明数据的时候到底都发现了什么细节。

数据声明

如果变量与执行上下文相关，那变量自己应该知道它的数据存储在哪里，并且知道如何访问。这种机制称为变量对象(variable object)。

变量对象(缩写为VO)是一个与执行上下文相关的特殊对象，它存储着在上下文中声明的以下内容：

```
变量 (var, 变量声明);  
函数声明 (FunctionDeclaration, 缩写为FD);  
函数的形参
```

举例来说，我们可以用普通的ECMAScript对象来表示一个变量对象：

```
VO = {};
```

就像我们所说的，VO就是执行上下文的属性(property)：

```
activeExecutionContext = {  
  VO: {  
    // 上下文数据 (var, FD, function arguments)  
  }  
};
```

只有全局上下文的变量对象允许通过VO的属性名称来间接访问(因为在全局上下文里，全局对象自身就是变量对象，稍后会详细介绍)，在其它上下文中是不能直接访问VO对象的，因为它只是内部机制的一个实现。

当我们声明一个变量或一个函数的时候，和我们创建VO新属性时一样没有别的区别（即：有名称以及对应的值）。

例如：

```
var a = 10;  
  
function test(x) {  
  var b = 20;  
};  
  
test(30);
```

对应的变量对象是：

```
// 全局上下文的变量对象
VO(globalContext) = {
  a: 10,
  test: function>
};

// test函数上下文的变量对象
VO(test functionContext) = {
  x: 30,
  b: 20
};
```

在具体实现层面(以及规范中)变量对象只是一个抽象概念。(从本质上说，在具体执行上下文中，VO名称是不一样的，并且初始结构也不一样。

不同执行上下文中的变量对象

对于所有类型的执行上下文来说，变量对象的一些操作(如变量初始化)和行为都是共通的。从这个角度来看，把变量对象作为抽象的基本事物来理解更为容易。同样在函数上下文中也定义和变量对象相关的额外内容。

抽象变量对象VO（变量初始化过程的一般行为）

```
├──> 全局上下文变量对象GlobalContextVO
      (VO === this === global)
└──> 函数上下文变量对象FunctionContextVO
      (VO === AO, 并且添加了<arguments>和<formal parameters>)
```

我们来详细看一下：

全局上下文中的变量对象

首先，我们要给全局对象一个明确的定义

全局对象(Global object) 是在进入任何执行上下文之前就已经创建了的对象；
这个对象只存在一份，它的属性在程序中任何地方都可以访问，全局对象的生命周期终止于程序退出那一刻。

全局对象初始创建阶段将Math、String、Date、parseInt作为自身属性，等属性初始化，同

样也可以有额外创建的其它对象作为属性（其可以指向到全局对象自身）。例如，在DOM中，全局对象的window属性就可以引用全局对象自身(当然，并不是所有的具体实现都是这样)：

```
global = {
  Math: <...>,
  String: <...>
  ...
  ...
  window: global //引用自身
};
```

当访问全局对象的属性时通常会忽略掉前缀，这是因为全局对象是不能通过名称直接访问的。不过我们依然可以通过全局上下文的this来访问全局对象，同样也可以递归引用自身。例如，DOM中的window。综上所述，代码可以简写为：

```
String(10); // 就是global.String(10);

// 带有前缀
window.a = 10; // === global.window.a = 10 === global.a = 10;
this.b = 20; // global.b = 20;
```

因此，回到全局上下文中的变量对象——在这里，变量对象就是全局对象自己：

```
VO(globalContext) === global;
```

非常有必要要理解上述结论，基于这个原理，在全局上下文中声明的对应，我们才可以间接通过全局对象的属性来访问它（例如，事先不知道变量名称）。

```
var a = new String('test');

alert(a); // 直接访问，在VO(globalContext)里找到："test"

alert(window['a']); // 间接通过global访问：global === VO(globalContext): "test"
alert(a === this.a); // true

var aKey = 'a';
alert(window[aKey]); // 间接通过动态属性名称访问："test"
```

函数上下文中的变量对象

在函数执行上下文中，VO是不能直接访问的，此时由活动对象(activation object,缩写为

AO)扮演VO的角色。

```
VO(functionContext) === AO;
```

活动对象是在进入函数上下文时刻被创建的，它通过函数的arguments属性初始化。arguments属性的值是Arguments对象：

```
AO = {  
  arguments: <Arg0>  
};
```

Arguments对象是活动对象的一个属性，它包括如下属性：

1. callee — 指向当前函数的引用
2. length — 真正传递的参数个数
3. properties-indexes (字符串类型的整数) 属性的值就是函数的参数值(按参数列表从左到右排列)。properties-indexes内部元素的个数等于arguments.length. properties-indexes 的值和实际传递进来的参数之间是共享的。

例如：

```
function foo(x, y, z) {  
  // 声明的函数参数数量arguments (x, y, z)  
  alert(foo.length); // 3  
  
  // 真正传进来的参数个数(only x, y)  
  alert(arguments.length); // 2  
  
  // 参数的callee是函数自身  
  alert(arguments.callee === foo); // true  
  
  // 参数共享  
  
  alert(x === arguments[0]); // true  
  alert(x); // 10  
  
  arguments[0] = 20;  
  alert(x); // 20  
  
  x = 30;  
  alert(arguments[0]); // 30  
  
  // 不过，没有传进来的参数z，和参数的第3个索引值是不共享的
```

```
    z = 40;
    alert(arguments[2]); // undefined

    arguments[2] = 50;
    alert(z); // 40
}

foo(10, 20);
```

这个例子的代码，在当前版本的Google Chrome浏览器里有一个bug — 即使没有传递参数 `z`，`z`和`arguments[2]`仍然是共享的。

处理上下文代码的2个阶段

现在我们终于到了本文的核心点了。执行上下文的代码被分成两个基本的阶段来处理：

1. 进入执行上下文
2. 执行代码

变量对象的修改变化与这两个阶段紧密相关。

注：这2个阶段的处理是一般行为，和上下文的类型无关（也就是说，在全局上下文和函数上下文中的表现是一样的）。

进入执行上下文

当进入执行上下文(代码执行之前)时，VO里已经包含了下列属性(前面已经说了)：

函数的所有形参(如果我們是在函数执行上下文中)

— 由名称和对应值组成的一个变量对象的属性被创建；没有传递对应参数的话，那么由名称和`undefined`值组成的一种变量对象的属性也将被创建。

所有函数声明(FunctionDeclaration, FD)

—由名称和对应值（函数对象(function-object)）组成一个变量对象的属性被创建；如果变量对象已经存在相同名称的属性，则完全替换这个属性。

所有变量声明(var, VariableDeclaration)

— 由名称和对应值（`undefined`）组成一个变量对象的属性被创建；如果变量名称跟已经声明的形式参数或函数相同，则变量声明不会干扰已经存在的这类属性。

让我们看一个例子：

```
function test(a, b) {
  var c = 10;
  function d() {}
  var e = function _e() {};
  (function x() {});
}

test(10); // call
```

当进入带有参数10的test函数上下文时，AO表现为如下：

```
AO(test) = {
  a: 10,
  b: undefined,
  c: undefined,
  d: <reference to FunctionDeclaration "d">
  e: undefined
};
```

注意，AO里并不包含函数“x”。这是因为“x”是一个函数表达式(FunctionExpression, 缩写为 FE) 而不是函数声明，函数表达式不会影响VO。不管怎样，函数“_e”同样也是函数表达式，但是就像我们下面将看到的那样，因为它分配给了变量“e”，所以它可以通过名称“e”来访问。函数声明FunctionDeclaration与函数表达式FunctionExpression的不同，将在第15章Functions进行详细的探讨，也可以参考本系列第2章[揭秘命名函数表达式](#)来了解。

这之后，将进入处理上下文代码的第二个阶段 — 执行代码。

代码执行

这个周期内，AO/VO已经拥有了属性(不过，并不是所有的属性都有值，大部分属性的值还是系统默认的初始值undefined)。

还是前面那个例子, AO/VO在代码解释期间被修改如下：

```
AO['c'] = 10;
AO['e'] = <reference to FunctionExpression "_e">;
```

再次注意，因为FunctionExpression“_e”保存到了已声明的变量“e”上，所以它仍然存在于内存中。而FunctionExpression“x”却不存在于AO/VO中，也就是说如果我们想尝试

调用 “x” 函数，不管在函数定义之前还是之后，都会出现一个错误 “x is not defined” ，未保存的函数表达式只有在它自己的定义或递归中才能被调用。

另一个经典例子：

```
alert(x); // function

var x = 10;
alert(x); // 10

x = 20;

function x() {};

alert(x); // 20
```

为什么第一个alert “x” 的返回值是function，而且它还是在 “x” 声明之前访问的 “x” 的？为什么不是10或20呢？因为，根据规范函数声明是在当进入上下文时填入的；同意周期，在进入上下文的时候还有一个变量声明 “x”，那么正如我们在上一个阶段所说，变量声明在顺序上跟在函数声明和形式参数声明之后，而且在这个进入上下文阶段，变量声明不会干扰VO中已经存在的同名函数声明或形式参数声明，因此，在进入上下文时，VO的结构如下：

```
VO = {};

VO['x'] = <reference to FunctionDeclaration "x">

// 找到var x = 10;
// 如果function "x"没有已经声明的话
// 这时候"x"的值应该是undefined
// 但是这个case里变量声明没有影响同名的function的值

VO['x'] = <the value is not disturbed, still function>
```

紧接着，在执行代码阶段，VO做如下修改：

```
VO['x'] = 10;
VO['x'] = 20;
```

我们可以在第二、三个alert看到这个效果。

在下面的例子里我们可以再次看到，变量是在进入上下文阶段放入VO中的。(因为，虽然else部分代码永远不会执行，但是不管怎样，变量 “b” 仍然存在于VO中。)


```

if (true) {
  var a = 1;
} else {
  var b = 2;
}

alert(a); // 1
alert(b); // undefined, 不是b没有声明, 而是b的值是undefined

```

关于变量

通常, 各类文章和JavaScript相关的书籍都声称: “不管是使用var关键字(在全局上下文)还是不使用var关键字(在任何地方), 都可以声明一个变量”。请记住, 这是错误的概念:

任何时候, 变量只能通过使用var关键字才能声明。

上面的赋值语句:

```
a = 10;
```

这仅仅是给全局对象创建了一个新属性(但它不是变量)。“不是变量”并不是说它不能被改变, 而是指它不符合ECMAScript规范中的变量概念, 所以它“不是变量”(它之所以能成为全局对象的属性, 完全是因为VO(globalContext) === global, 大家还记得这个吧?)。

让我们通过下面的实例看看具体的区别吧:

```

alert(a); // undefined
alert(b); // "b" 没有声明

b = 10;
var a = 20;

```

所有根源仍然是VO和进入上下文阶段和代码执行阶段:

进入上下文阶段:

```

VO = {
  a: undefined
};

```

我们可以看到, 因为“b”不是一个变量, 所以在这个阶段根本就没有“b”, “b”将只在代码执行阶段才会出现(但是在我们这个例子里, 还没有到那就已经出错了)。

让我们改变一下例子代码：

```
alert(a); // undefined, 这个大家都知道,
b = 10;
alert(b); // 10, 代码执行阶段创建
var a = 20;
alert(a); // 20, 代码执行阶段修改
```

关于变量，还有一个重要的知识点。变量相对于简单属性来说，变量有一个特性 (attribute) : {DontDelete},这个特性的含义就是不能用delete操作符直接删除变量属性。

```
a = 10;
alert(window.a); // 10

alert(delete a); // true

alert(window.a); // undefined

var b = 20;
alert(window.b); // 20

alert(delete b); // false

alert(window.b); // still 20
```

但是这个规则在有个上下文里不起走样，那就是eval上下文，变量没有{DontDelete}特性。

```
eval('var a = 10;');
alert(window.a); // 10

alert(delete a); // true

alert(window.a); // undefined
```

使用一些调试工具(例如：Firebug)的控制台测试该实例时，请注意，Firebug同样也是使用eval来执行控制台里你的代码。因此，变量属性同样没有{DontDelete}特性，可以被删除。

特殊实现: parent 属性

前面已经提到过，按标准规范，活动对象是不可能被直接访问到的。但是，一些具体实现并没有完全遵守这个规定，例如SpiderMonkey和Rhino；的实现中，函数有一个特殊的属性parent，通过这个属性可以直接引用到活动对象（或全局变量对象），在此对象里创建了函数。

例如 (SpiderMonkey, Rhino) :

```
var global = this;
var a = 10;

function foo() {}

alert(foo.__parent__); // global

var V0 = foo.__parent__;

alert(V0.a); // 10
alert(V0 === global); // true
```

在上面的例子中我们可以看到，函数foo是在全局上下文中创建的，所以属性parent 指向全局上下文的变量对象，即全局对象。

然而，在SpiderMonkey中用同样的方式访问活动对象是不可能的：在不同版本的SpiderMonkey中，内部函数的parent 有时指向null ，有时指向全局对象。

在Rhino中，用同样的方式访问活动对象是完全可以的。

例如 (Rhino) :

```
var global = this;
var x = 10;

(function foo() {

    var y = 20;

    // "foo"上下文里的活动对象
    var A0 = (function () {}).__parent__;

    print(A0.y); // 20

    // 当前活动对象的__parent__ 是已经存在的全局对象
    // 变量对象的特殊链形成了
    // 所以我们叫做作用域链
    print(A0.__parent__ === global); // true

    print(A0.__parent__.x); // 10

})();
```

总结

在这篇文章里，我们深入学习了跟执行上下文相关的对象。我希望这些知识对您来说能有所

帮助，能解决一些您曾经遇到的问题或困惑。按照计划，在后续的章节中，我们将探讨作用域链，标识符解析，闭包。

有任何问题，我很高兴在下面评论中能帮你解答。

其它参考

- 10.1.3 – [Variable Instantiation](#);
- 10.1.5 – [Global Object](#);
- 10.1.6 – [Activation Object](#);
- 10.1.8 – [Arguments Object](#).

(13) This? Yes, this!

介绍

在这篇文章里，我们将讨论跟执行上下文直接相关的更多细节。讨论的主题就是this关键字。实践证明，这个主题很难，在不同执行上下文中this的确定经常会发生问题。

许多程序员习惯的认为，在程序语言中，this关键字与面向对象程序开发紧密相关，其完全指向由构造器新创建的对象。在ECMAScript规范中也是这样实现的，但正如我们将看到那样，在ECMAScript中，this并不限于只用来指向新创建的对象。

英文翻译: Dmitry A. Soshnikov在Stoyan Stefanov的帮助下

发布: 2010-03-07

<http://dmitrysoshnikov.com/ecmascript/chapter-3-this/>

俄文原文: Dmitry A. Soshnikov

修正: Zeroglif

发布: 2009-06-28;

更新: 2010-03-07

<http://dmitrysoshnikov.com/ecmascript/ru-chapter-3-this/>

本文绝大部分内容参考了: <http://www.denisdeng.com/?p=900>

部分句子参考了: [justin的中文翻译](#)

让我们更详细的了解一下，在ECMAScript中this到底是什么？

定义

this是执行上下文中的一个属性：

```
activeExecutionContext = {  
  VO: {...},  
  this: thisValue  
};
```

这里VO是我们前一章讨论的变量对象。

this与上下文中可执行代码的类型有直接关系，this值在进入上下文时确定，并且在上下文运行期间永久不变。

下面让我们更详细研究这些案例：

全局代码中的this

在这里一切都简单。在全局代码中，this始终是全局对象本身，这样就有可能间接的引用到它了。

```
// 显示定义全局对象的属性
this.a = 10; // global.a = 10
alert(a); // 10

// 通过赋值给一个无标示符隐式
b = 20;
alert(this.b); // 20

// 也是通过变量声明隐式声明的
// 因为全局上下文的变量对象是全局对象自身
var c = 30;
alert(this.c); // 30
```

函数代码中的this

在函数代码中使用this时很有趣，这种情况很难且会导致很多问题。

这种类型的代码中，this值的首要特点（或许是最主要的）是它不是静态的绑定到一个函数。

正如我们上面曾提到的那样，this是进入上下文时确定，在一个函数代码中，这个值在每一次完全不同。

不管怎样，在代码运行时的this值是不变的，也就是说，因为它不是一个变量，就不可能为其分配一个新值（相反，在Python编程语言中，它明确的定义为对象本身，在运行期间可以不断改变）。

```
var foo = {x: 10};

var bar = {
  x: 20,
  test: function () {

    alert(this === bar); // true
    alert(this.x); // 20
  }
}
```

```
this = foo; // 错误, 任何时候不能改变this的值

alert(this.x); // 如果不出错的话, 应该是10, 而不是20

}

};

// 在进入上下文的时候
// this被当成bar对象
// determined as "bar" object; why so - will
// be discussed below in detail

bar.test(); // true, 20

foo.test = bar.test;

// 不过, 这里this依然不会是foo
// 尽管调用的是相同的function

foo.test(); // false, 10
```

那么, 影响了函数代码中this值的变化有几个因素:

首先, 在通常的函数调用中, this是由激活上下文代码的调用者来提供的, 即调用函数的父上下文(parent context)。this取决于调用函数的方式。

为了在任何情况下准确无误的确定this值, 有必要理解和记住这重要的一点。正是调用函数的方式影响了调用的上下文中的this值, 没有别的什么(我们可以在一些文章, 甚至是在关于javascript的书籍中看到, 它们声称: “this值取决于函数如何定义, 如果它是全局函数, this设置为全局对象, 如果函数是一个对象的方法, this将总是指向这个对象。-这绝对不正确”)。继续我们的话题, 可以看到, 即使是正常的全局函数也会被调用方式的不同形式激活, 这些不同的调用方式导致了不同的this值。

```
function foo() {
    alert(this);
}

foo(); // global

alert(foo === foo.prototype.constructor); // true

// 但是同一个function的不同的调用表达式, this是不同的

foo.prototype.constructor(); // foo.prototype
```

有可能作为一些对象定义的方法来调用函数, 但是this将不会设置为这个对象。

```
var foo = {
  bar: function () {
    alert(this);
    alert(this === foo);
  }
};

foo.bar(); // foo, true

var exampleFunc = foo.bar;

alert(exampleFunc === foo.bar); // true

// 再一次，同一个function的不同的调用表达式，this是不同的

exampleFunc(); // global, false
```

那么，调用函数的方式如何影响this值？为了充分理解this值的确定，需要详细分析其内部类型之一——引用类型（Reference type）。

引用类型（Reference type）

使用伪代码我们可以将引用类型的值可以表示为拥有两个属性的对象——base（即拥有属性的那个对象），和base中的propertyName。

```
var valueOfReferenceType = {
  base: <base object>,
  propertyName: <property name>
};
```

引用类型的值只有两种情况：

1. 当我们处理一个标示符时
2. 或一个属性访问器

标示符的处理过程在下一篇文章里详细讨论，在这里我们只需要知道，在该算法的返回值中，总是一个引用类型的值（这对this来说很重要）。

标识符是变量名，函数名，函数参数名和全局对象中未识别的属性名。例如，下面标识符的值：

```
var foo = 10;
function bar() {}
```


在操作的中间结果中，引用类型对应的值如下：

```
var fooReference = {
  base: global,
  propertyName: 'foo'
};

var barReference = {
  base: global,
  propertyName: 'bar'
};
```

为了从引用类型中得到一个对象真正的值，伪代码中的GetValue方法可以做如下描述：

```
function GetValue(value) {
  if (Type(value) !== Reference) {
    return value;
  }

  var base = GetBase(value);

  if (base === null) {
    throw new ReferenceError;
  }

  return base.[[Get]](GetPropertyname(value));
}
```

内部的[[Get]]方法返回对象属性真正的值，包括对原型链中继承的属性分析。

```
GetValue(fooReference); // 10
GetValue(barReference); // function object "bar"
```

属性访问器都应该熟悉。它有两种变体：点（.）语法（此时属性名是正确的标示符，且事先知道），或括号语法（[]）。

```
foo.bar();
foo['bar']();
```

在中间计算的返回值中，我们有了引用类型的值。

```
var fooBarReference = {
```

```
    base: foo,
    propertyName: 'bar'
  };

  GetValue(fooBarReference); // function object "bar"
```

引用类型的值与函数上下文中的this值如何相关？——从最重要的意义上来说。这个关联的过程是这篇文章的核心。一个函数上下文中确定this值的通用规则如下：

在一个函数上下文中，this由调用者提供，由调用函数的方式来决定。如果调用括号()的左边是引用类型的值，this将设为引用类型值的base对象（base object），在其他情况下（与引用类型不同的任何其它属性），这个值为null。不过，实际不存在this的值为null的情况，因为当this的值为null的时候，其值会被隐式转换为全局对象。注：第5版的ECMAScript中，已经不强迫转换成全局变量了，而是赋值为undefined。

我们看看这个例子中的表现：

```
function foo() {
  return this;
}

foo(); // global
```

我们看到在调用括号的左边是一个引用类型值（因为foo是一个标示符）。

```
var fooReference = {
  base: global,
  propertyName: 'foo'
};
```

相应地，this也设置为引用类型的base对象。即全局对象。

同样，使用属性访问器：

```
var foo = {
  bar: function () {
    return this;
  }
};

foo.bar(); // foo
```

我们再次拥有一个引用类型，其base是foo对象，在函数bar激活时用作this。

```
var fooBarReference = {  
  base: foo,  
  propertyName: 'bar'  
};
```

但是，用另外一种形式激活相同的函数，我们得到其它的this值。

```
var test = foo.bar;  
test(); // global
```

因为test作为标示符，生成了引用类型的其他值，其base（全局对象）用作this值。

```
var testReference = {  
  base: global,  
  propertyName: 'test'  
};
```

现在，我们可以很明确的告诉你，为什么用表达式的不同形式激活同一个函数会不同的this值，答案在于引用类型（type Reference）不同的中间值。

```
function foo() {  
  alert(this);  
}  
  
foo(); // global, because  
  
var fooReference = {  
  base: global,  
  propertyName: 'foo'  
};  
  
alert(foo === foo.prototype.constructor); // true  
  
// 另外一种形式的调用表达式  
  
foo.prototype.constructor(); // foo.prototype, because  
  
var fooPrototypeConstructorReference = {  
  base: foo.prototype,  
  propertyName: 'constructor'  
};
```

另外一个通过调用方式动态确定this值的经典例子：

```
function foo() {  
    alert(this.bar);  
}  
  
var x = {bar: 10};  
var y = {bar: 20};  
  
x.test = foo;  
y.test = foo;  
  
x.test(); // 10  
y.test(); // 20
```

函数调用和非引用类型

因此，正如我们已经指出，当调用括号的左边不是引用类型而是其它类型，这个值自动设置为null，结果为全局对象。

让我们再思考这种表达式：

```
(function () {  
    alert(this); // null => global  
})();
```

在这个例子中，我们有一个函数对象但不是引用类型的对象（它不是标示符，也不是属性访问器），相应地，this值最终设为全局对象。

更多复杂的例子：

```
var foo = {  
    bar: function () {  
        alert(this);  
    }  
};  
  
foo.bar(); // Reference, OK => foo  
(foo.bar)(); // Reference, OK => foo  
  
(foo.bar = foo.bar)(); // global?  
(false || foo.bar)(); // global?  
(foo.bar, foo.bar)(); // global?
```

为什么我们有一个属性访问器，它的中间值应该为引用类型的值，在某些调用中我们得到的this值不是base对象，而是global对象？

问题在于后面的三个调用，在应用一定的运算操作之后，在调用括号的左边的值不再是引用

本文档使用 [看云](#) 构建

类型。

1. 第一个例子很明显——明显的引用类型，结果是，this为base对象，即foo。
2. 在第二个例子中，组运算符并不适用，想想上面提到的，从引用类型中获得一个对象真正的值的方法，如GetValue。相应的，在组运算的返回中——我们得到仍是一个引用类型。这就是this值为什么再次设为base对象，即foo。
3. 第三个例子中，与组运算符不同，赋值运算符调用了GetValue方法。返回的结果是函数对象（但不是引用类型），这意味着this设为null，结果是global对象。
4. 第四个和第五个也是一样——逗号运算符和逻辑运算符（OR）调用了GetValue方法，相应地，我们失去了引用而得到了函数。并再次设为global。

引用类型和this为null

有一种情况是这样的：当调用表达式限定了call括号左边的引用类型的值，尽管this被设定为null，但结果被隐式转化成global。当引用类型值的base对象是被活动对象时，这种情况就会出现。

下面的实例中，内部函数被父函数调用，此时我们就能够看到上面说的那种特殊情况。正如我们在第12章知道的一样，局部变量、内部函数、形式参数储存在给定函数的激活对象中。

```
function foo() {
  function bar() {
    alert(this); // global
  }
  bar(); // the same as AO.bar()
}
```

活动对象总是作为this返回，值为null——（即伪代码的AO.bar()相当于null.bar()）。这里我们再次回到上面描述的例子，this设置为全局对象。

有一种情况除外：如果with对象包含一个函数名属性，在with语句的内部块中调用函数。With语句添加到该对象作用域的最前端，即在活动对象的前面。相应地，也就有了引用类型（通过标示符或属性访问器），其base对象不再是活动对象，而是with语句的对象。顺便提一句，它不仅与内部函数相关，也与全局函数相关，因为with对象比作用域链里的最前端的对象(全局对象或一个活动对象)还要靠前。

```
var x = 10;

with ({
  foo: function () {
```

```

        alert(this.x);
    },
    x: 20
}) {

    foo(); // 20
}

// because

var fooReference = {
    base: __withObject,
    propertyName: 'foo'
};

```

同样的情况出现在catch语句的实际参数中函数调用：在这种情况下，catch对象添加到作用域的最前端，即在活动对象或全局对象的前面。但是，这个特定的行为被确认为ECMA-262-3的一个bug，这个在新版的ECMA-262-5中修复了。这样，在特定的活动对象中，this指向全局对象。而不是catch对象。

```

try {
    throw function () {
        alert(this);
    };
} catch (e) {
    e(); // ES3标准里是__catchObject, ES5标准里是global
}

// on idea

var eReference = {
    base: __catchObject,
    propertyName: 'e'
};

// ES5新标准里已经fix了这个bug,
// 所以this就是全局对象了
var eReference = {
    base: global,
    propertyName: 'e'
};

```

同样的情况出现在命名函数（函数的更对细节参考第15章Functions）的递归调用中。在函数的第一次调用中，base对象是父活动对象（或全局对象），在递归调用中，base对象应该是存储着函数表达式可选名称的特定对象。但是，在这种情况下，this总是指向全局对象。

```
(function foo(bar) {  
    alert(this);  
  
    !bar && foo(1); // "should" be special object, but always (correct) global  
  
})(); // global
```

作为构造器调用的函数中的this

还有一个与this值相关的情况是在函数的上下文中，这是一个构造函数的调用。

```
function A() {  
    alert(this); // "a"对象下创建一个新属性  
    this.x = 10;  
}  
  
var a = new A();  
alert(a.x); // 10
```

在这个例子中，new运算符调用“A”函数的内部的[[Construct]]方法，接着，在对象创建后，调用内部的[[Call]]方法。所有相同的函数“A”都将this的值设置为新创建的对象。

函数调用中手动设置this

在函数原型中定义的两个方法（因此所有的函数都可以访问它）允许去手动设置函数调用的this值。它们是.apply和.call方法。他们用接受的第一个参数作为this值，this在调用的作用域中使用。这两个方法的区别很小，对于.apply，第二个参数必须是数组（或者是类似数组的对象，如arguments，反过来，.call能接受任何参数。两个方法必须的参数是第一个——this。

例如：

```
var b = 10;  
  
function a(c) {  
    alert(this.b);  
    alert(c);  
}  
  
a(20); // this === global, this.b == 10, c == 20  
  
a.call({b: 20}, 30); // this === {b: 20}, this.b == 20, c == 30  
a.apply({b: 30}, [40]) // this === {b: 30}, this.b == 30, c == 40
```

结论

在这篇文章中，我们讨论了ECMAScript中this关键字的特征（对比于C++ 和 Java，它们的确是特色）。我希望这篇文章有助于你准确的理解ECMAScript中this关键字如何工作。同样，我很乐意在评论中回到你的问题。

其它参考

- 10.1.7 – [This](#)
- 11.1.1 – [The this keyword](#)
- 11.2.2 – [The new operator](#)
- 11.2.3 – [Function calls](#)

(14) 作用域链(Scope Chain)

前言

在第12章关于变量对象的描述中，我们已经知道一个执行上下文 的数据（变量、函数声明和函数的形参）作为属性存储在变量对象中。

同时我们也知道变量对象在每次进入上下文时创建，并填入初始值，值的更新出现在代码执行阶段。

这一章专门讨论与执行上下文直接相关的更多细节，这次我们将提及一个议题——作用域链。

英文原文：<http://dmitrysoshnikov.com/ecmascript/chapter-4-scope-chain/>

中文参考：<http://www.denisdeng.com/?p=908>

本文绝大部分内容来自上述地址，仅做少许修改，感谢作者

定义

如果要简要的描述并展示其重点，那么作用域链大多数与内部函数相关。

我们知道，ECMAScript 允许创建内部函数，我们甚至能从父函数中返回这些函数。

```
var x = 10;

function foo() {
  var y = 20;
  function bar() {
    alert(x + y);
  }
  return bar;
}

foo(); // 30
```

这样，很明显每个上下文拥有自己的变量对象：对于全局上下文，它是全局对象自身；对于函数，它是活动对象。

作用域链正是内部上下文所有变量对象（包括父变量对象）的列表。此链用来变量查询。即上面的例子中，“bar”上下文的作用域链包括AO(bar)、AO(foo)和VO(global)。

但是，让我们仔细研究这个问题。

让我们从定义开始，并进一步的讨论示例。

作用域链与一个执行上下文相关，变量对象的链用于在标识符解析中变量查找。

函数上下文的作用域链在函数调用时创建的，包含活动对象和这个函数内部的[[scope]]属性。下面我们将更详细的讨论一个函数的[[scope]]属性。

在上下文中示意如下：

```
activeExecutionContext = {
  VO: {...}, // or AO
  this: thisValue,
  Scope: [ // Scope chain
    // 所有变量对象的列表
    // for identifiers lookup
  ]
};
```

其scope定义如下：

```
Scope = AO + [[Scope]]
```

这种联合和标识符解析过程，我们将在下面讨论，这与函数的生命周期相关。

函数的生命周期

函数的生命周期分为创建和激活阶段（调用时），让我们详细研究它。

函数创建

众所周知，在进入上下文时函数声明放到变量/活动（VO/AO）对象中。让我们看看在全局上下文中的变量和函数声明（这里变量对象是全局对象自身，我们还记得，是吧？）

```
var x = 10;

function foo() {
  var y = 20;
  alert(x + y);
}

foo(); // 30
```

在函数激活时，我们得到正确的（预期的）结果 - - 30。但是，有一个很重要的特点。

本文档使用 [看云](#) 构建

此前，我们仅仅谈到有关当前上下文的变量对象。这里，我们看到变量“y”在函数“foo”中定义（意味着它在foo上下文的AO中），但是变量“x”并未在“foo”上下文中定义，相应地，它也不会添加到“foo”的AO中。乍一看，变量“x”相对于函数“foo”根本就不存在；但正如我们在下面看到的——也仅仅是“一瞥”，我们发现，“foo”上下文的活动对象中仅包含一个属性——“y”。

```
fooContext.AO = {
  y: undefined // undefined - 进入上下文的时候是20 - at activation
};
```

函数“foo”如何访问到变量“x”？理论上函数应该能访问一个更高一层上下文的变量对象。实际上它正是这样，这种机制是通过函数内部的[[scope]]属性来实现的。

[[scope]]是所有父变量对象的层级链，处于当前函数上下文之上，在函数创建时存于其中。

注意这重要的一点——[[scope]]在函数创建时被存储——静态（不变的），永远永远，直至函数销毁。即：函数可以永不调用，但[[scope]]属性已经写入，并存储在函数对象中。

另外一个需要考虑的是——与作用域链对比，[[scope]]是函数的一个属性而不是上下文。考虑到上面的例子，函数“foo”的[[scope]]如下：

```
foo.[[Scope]] = [
  globalContext.VO // === Global
];
```

举例来说，我们用通常的ECMAScript 数组展现作用域和[[scope]]。

继续，我们知道在函数调用时进入上下文，这时候活动对象被创建，this和作用域（作用域链）被确定。让我们详细考虑这一时刻。

函数激活

正如在定义中说到的，进入上下文创建AO/VO之后，上下文的Scope属性（变量查找的一个作用域链）作如下定义：

```
Scope = AO|VO + [[Scope]]
```

上面代码的意思是：活动对象是作用域数组的第一个对象，即添加到作用域的前端。

```
Scope = [AO].concat([[Scope]]);
```

本文档使用 [看云](#) 构建

这个特点对于标示符解析的处理来说很重要。

标示符解析是一个处理过程，用来确定一个变量（或函数声明）属于哪个变量对象。

这个算法的返回值中，我们总有一个引用类型，它的base组件是相应的变量对象（或若未找到则为null），属性名组件是向上查找的标示符的名称。引用类型的详细信息在第13章.this中已讨论。

标识符解析过程包含与变量名对应属性的查找，即作用域中变量对象的连续查找，从最深的上下文开始，绕过作用域链直到最上层。

这样一来，在向上查找中，一个上下文中的局部变量较之于父作用域的变量拥有较高的优先级。万一两个变量有相同的名称但来自不同的作用域，那么第一个被发现的是在最深作用域中。

我们用一个稍微复杂的例子描述上面讲到的这些。

```
var x = 10;

function foo() {
  var y = 20;

  function bar() {
    var z = 30;
    alert(x + y + z);
  }

  bar();
}

foo(); // 60
```

对此，我们有如下的变量/活动对象，函数的[[scope]]属性以及上下文的作用域链：

全局上下文的变量对象是：

```
globalContext.VO === Global = {
  x: 10
  foo: function>
};
```

在“foo”创建时，“foo”的[[scope]]属性是：

```
foo.[[Scope]] = [  
  globalContext.V0  
];
```

在 “foo” 激活时（进入上下文），“foo” 上下文的活动对象是：

```
fooContext.A0 = {  
  y: 20,  
  bar: function >  
};
```

“foo” 上下文的作用域链为：

```
fooContext.Scope = fooContext.A0 + foo.[[Scope]] // i.e.:  
  
fooContext.Scope = [  
  fooContext.A0,  
  globalContext.V0  
];
```

内部函数 “bar” 创建时，其[[scope]]为：

```
bar.[[Scope]] = [  
  fooContext.A0,  
  globalContext.V0  
];
```

在 “bar” 激活时，“bar” 上下文的活动对象为：

```
barContext.A0 = {  
  z: 30  
};
```

“bar” 上下文的作用域链为：

```
barContext.Scope = barContext.A0 + bar.[[Scope]] // i.e.:  
  
barContext.Scope = [  
  barContext.A0,  
  fooContext.A0,  
  globalContext.V0  
];
```

对 “x” 、 “y” 、 “z” 的标识符解析如下：

```
- "x"
-- barContext.A0 // not found
-- fooContext.A0 // not found
-- globalContext.V0 // found - 10

- "y"
-- barContext.A0 // not found
-- fooContext.A0 // found - 20

- "z"
-- barContext.A0 // found - 30
```

作用域特征

让我们看看与作用域链和函数[[scope]]属性相关的一些重要特征。

闭包

在ECMAScript中，闭包与函数的[[scope]]直接相关，正如我们提到的那样，[[scope]]在函数创建时被存储，与函数共存亡。实际上，闭包是函数代码和其[[scope]]的结合。因此，作为其对象之一，[[Scope]]包括在函数内创建的词法作用域（父变量对象）。当函数进一步激活时，在变量对象的这个词法链（静态的存储于创建时）中，来自较高作用域的变量将被搜寻。

例如：

```
var x = 10;

function foo() {
  alert(x);
}

(function () {
  var x = 20;
  foo(); // 10, but not 20
})();
```

我们再次看到，在标识符解析过程中，使用函数创建时定义的词法作用域 - - 变量解析为 10，而不是30。此外，这个例子也清晰的表明，一个函数（这个例子中为从函数 “foo” 返回的匿名函数）的[[scope]]持续存在，即使是在函数创建的作用域已经完成之后。

关于ECMAScript中闭包的理论和其执行机制的更多细节，阅读16章闭包。

通过构造函数创建的函数的[[scope]]

本文档使用 [看云](#) 构建

在上面的例子中，我们看到，在函数创建时获得函数的[[scope]]属性，通过该属性访问到所有父上下文的变量。但是，这个规则有一个重要的例外，它涉及到通过函数构造函数创建的函数。

```
var x = 10;

function foo() {
    var y = 20;

    function barFD() { // 函数声明
        alert(x);
        alert(y);
    }

    var barFE = function () { // 函数表达式
        alert(x);
        alert(y);
    };

    var barFn = Function('alert(x); alert(y);');

    barFD(); // 10, 20
    barFE(); // 10, 20
    barFn(); // 10, "y" is not defined
}

foo();
```

我们看到，通过函数构造函数（Function constructor）创建的函数“bar”，是不能访问变量“y”的。但这并不意味着函数“barFn”没有[[scope]]属性（否则它不能访问到变量“x”）。问题在于通过函数构造函数创建的函数的[[scope]]属性总是唯一的全局对象。考虑到这一点，如通过这种函数创建除全局之外的最上层的上下文闭包是不可能的。

二维作用域链查找

在作用域链中查找最重要的一点是变量对象的属性（如果有的话）须考虑其中 - - 源于 ECMAScript 的原型特性。如果一个属性在对象中没有直接找到，查询将在原型链中继续。即常说的二维链查找。（1）作用域链环节；（2）每个作用域链 - - 深入到原型链环节。如果在Object.prototype中定义了属性，我们能看到这种效果。

```
function foo() {
    alert(x);
}

Object.prototype.x = 10;
```

```
foo(); // 10
```

活动对象没有原型，我们可以在下面的例子中看到：

```
function foo() {
    var x = 20;

    function bar() {
        alert(x);
    }

    bar();
}

Object.prototype.x = 10;

foo(); // 20
```

如果函数“bar”上下文的激活对象有一个原型，那么“x”将在Object.prototype中被解析，因为它在AO中不被直接解析。但在上面的第一个例子中，在标识符解析中，我们到达全局对象（在一些执行中并不全是这样），它从Object.prototype继承而来，响应地，“x”解析为10。

同样的情况出现在一些版本的SpiderMonkey 的命名函数表达式（缩写为NFE）中，在那里特定的对象存储从Object.prototype继承而来的函数表达式的可选名称，在Blackberry中的一些版本中，执行时激活对象从Object.prototype继承。但是，关于该特色的更多细节在第15章函数讨论。

全局和eval上下文中的作用域链

这里不一定很有趣，但必须要提示一下。全局上下文的作用域链仅包含全局对象。代码eval的上下文与当前的调用上下文（calling context）拥有同样的作用域链。

```
globalContext.Scope = [
    Global
];

evalContext.Scope === callingContext.Scope;
```

代码执行时对作用域链的影响

在ECMAScript 中，在代码执行阶段有两个声明能修改作用域链。这就是with声明和catch语句。它们添加到作用域链的最前端，对象须在这些声明中出现的标识符中查找。如果发生其

中的一个，作用域链简要的作如下修改：

```
Scope = withObject|catchObject + A0|V0 + [[Scope]]
```

在这个例子中添加对象，对象是它的参数（这样，没有前缀，这个对象的属性变得可以访问）。

```
var foo = {x: 10, y: 20};

with (foo) {
  alert(x); // 10
  alert(y); // 20
}
```

作用域链修改成这样：

```
Scope = foo + A0|V0 + [[Scope]]
```

我们再次看到，通过with语句，对象中标识符的解析添加到作用域链的最前端：

```
var x = 10, y = 10;

with ({x: 20}) {
  var x = 30, y = 30;

  alert(x); // 30
  alert(y); // 30
}

alert(x); // 10
alert(y); // 30
```

在进入上下文时发生了什么？标识符“x”和“y”已被添加到变量对象中。此外，在代码运行阶段作如下修改：

1. x = 10, y = 10;
2. 对象{x:20}添加到作用域的前端;
3. 在with内部，遇到了var声明，当然什么也没创建，因为在进入上下文时，所有变量已被解析添加;
4. 在第二步中，仅修改变量“x”，实际上对象中的“x”现在被解析，并添加到作用域链

的最前端，“x”为20，变为30;

5. 同样也有变量对象“y”的修改，被解析后其值也相应的由10变为30;
6. 此外，在with声明完成后，它的特定对象从作用域链中移除（已改变的变量“x” - - 30也从那个对象中移除），即作用域链的结构恢复到with得到加强以前的状态。
7. 在最后两个alert中，当前变量对象的“x”保持同一，“y”的值现在等于30，在with声明运行中已发生改变。

同样，catch语句的异常参数变得可以访问，它创建了只有一个属性的新对象 - - 异常参数名。图示看起来像这样：

```
try {
  ...
} catch (ex) {
  alert(ex);
}
```

作用域链修改为：

```
var catchObject = {
  ex:
};

Scope = catchObject + A0|V0 + [[Scope]]
```

在catch语句完成运行之后，作用域链恢复到以前的状态。

结论

在这个阶段，我们几乎考虑了与执行上下文相关的所有常用概念，以及与它们相关的细节。按照计划 - - 函数对象的详细分析：函数类型（函数声明，函数表达式）和闭包。顺便说一下，在这篇文章中，闭包直接与[[scope]]属性相关，但是，关于它将在合适的篇章中讨论。我很乐意在评论中回答你的问题。

其它参考

- 8.6.2 - [\[\[Scope\]\]](#)
- 10.1.4 - [Scope Chain and Identifier Resolution](#)

(15) 函数 (Functions)

介绍

本章节我们要着重介绍的是一个非常常见的ECMAScript对象——函数 (function),我们将详细讲解一下各种类型的函数是如何影响上下文的变量对象以及每个函数的作用域链都包含什么,以及回答诸如像下面这样的问题:下面声明的函数有什么区别么?(如果有,区别是什么)。

原文: <http://dmitrysoshnikov.com/ecmascript/chapter-5-functions/>

```
var foo = function () {  
    ...  
};
```

平时的惯用方式:

```
function foo() {  
    ...  
}
```

或者,下面的函数为什么要用括号括住?

```
(function () {  
    ...  
})();
```

关于具体的介绍,早前面的[12章变量对象](#)和[14章作用域链](#)都有介绍,如果需要详细了解这些内容,请查询上述2个章节的详细内容。

但我们依然要一个一个分别看看,首先从函数的类型讲起:

函数类型

在ECMAScript 中有三种函数类型:函数声明,函数表达式和函数构造器创建的函数。每一种都有自己的特点。

函数声明

函数声明 (缩写为FD) 是这样一种函数:

1. 有一个特定的名称
2. 在源码中的位置：要么处于程序级 (Program level) ，要么处于其它函数的主体 (FunctionBody) 中
3. 在进入上下文阶段创建
4. 影响变量对象
5. 以下面的方式声明

```
function exampleFunc() {  
    ...  
}
```

这种函数类型的主要特点在于它们仅仅影响变量对象 (即存储在上下文的VO中的变量对象) 。该特点也解释了第二个重要点 (它是变量对象特性的结果) ——在代码执行阶段它们已经可用 (因为FD在进入上下文阶段已经存在于VO中——代码执行之前) 。

例如 (函数在其声明之前被调用)

```
foo();  
  
function foo() {  
    alert('foo');  
}
```

另外一个重点知识点是上述定义中的第二点——函数声明在源码中的位置：

```
// 函数可以在如下地方声明：  
// 1) 直接在全局上下文中  
function globalFD() {  
    // 2) 或者在一个函数的函数体内  
    function innerFD() {}  
}
```

只有这2个位置可以声明函数，也就是说:不可能在表达式位置或一个代码块中定义它。

另外一种可以取代函数声明的方式是函数表达式，解释如下：

函数表达式

函数表达式 (缩写为FE) 是这样一种函数：

1. 在源码中须出现在表达式的位置
2. 有可选的名称

3. 不会影响变量对象
4. 在代码执行阶段创建

这种函数类型的主要特点在于它在源码中总是处在表达式的位置。最简单的一个例子就是一个赋值声明：

```
var foo = function () {
  ...
};
```

该例演示是让一个匿名函数表达式赋值给变量foo，然后该函数可以用foo这个名称进行访问——foo()。

同时和定义里描述的一样，函数表达式也可以拥有可选的名称：

```
var foo = function _foo() {
  ...
};
```

需要注意的是，在外部FE通过变量“foo”来访问——foo()，而在函数内部（如递归调用），有可能使用名称“_foo”。

如果FE有一个名称，就很难与FD区分。但是，如果你明白定义，区分起来就简单明了：FE总是处在表达式的位置。在下面的例子中我们可以看到各种ECMAScript 表达式：

```
// 圆括号（分组操作符）内只能是表达式
(function foo() {});

// 在数组初始化器内只能是表达式
[function bar() {}];

// 逗号也只能操作表达式
1, function baz() {};
```

表达式定义里说明：FE只能在代码执行阶段创建而且不存在于变量对象中，让我们来看一个示例行为：

```
// FE在定义阶段之前不可用（因为它是在代码执行阶段创建）

alert(foo); // "foo" 未定义

(function foo() {});
```

```
// 定义阶段之后也不可用，因为他不在变量对象VO中

alert(foo); // "foo" 未定义
```

相当一部分问题出现了，我们为什么需要函数表达式？答案很明显——在表达式中使用它们，“不会污染”变量对象。最简单的例子是将一个函数作为参数传递给其它函数。

```
function foo(callback) {
    callback();
}

foo(function bar() {
    alert('foo.bar');
});

foo(function baz() {
    alert('foo.baz');
});
```

在上述例子里，FE赋值给了一个变量（也就是参数），函数将该表达式保存在内存中，并通过变量名来访问（因为变量影响变量对象），如下：

```
var foo = function () {
    alert('foo');
};

foo();
```

另外一个例子是创建封装的闭包从外部上下文中隐藏辅助性数据（在下面的例子中我们使用FE，它在创建后立即调用）：

```
var foo = {};

(function initialize() {
    var x = 10;

    foo.bar = function () {
        alert(x);
    };
})();

foo.bar(); // 10;

alert(x); // "x" 未定义
```

我们看到函数foo.bar (通过[[Scope]]属性) 访问到函数initialize的内部变量 “x” 。同时, “x” 在外部不能直接访问。在许多库中, 这种策略常用来创建“私有”数据和隐藏辅助实体。在这种模式中, 初始化的FE的名称通常被忽略:

```
(function () {
  // 初始化作用域
})();
```

还有一个例子是: 在代码执行阶段通过条件语句进行创建FE, 不会污染变量对象VO。

```
var foo = 10;

var bar = (foo % 2 == 0
  ? function () { alert(0); }
  : function () { alert(1); }
);

bar(); // 0
```

关于圆括号的问题

让我们回头并回答在文章开头提到的问题——“为何在函数创建后的立即调用中必须用圆括号来包围它?”, 答案就是: 表达式句子的限制就是这样的。

按照标准, 表达式语句不能以一个大括号(开始是因为他很难与代码块区分, 同样, 他也不能以函数关键字开始, 因为很难与函数声明进行区分。即, 所以, 如果我们定义一个立即执行的函数, 在其创建后立即按以下方式调用:

```
function () {
  ...
}();

// 即便有名称

function foo() {
  ...
}();
```

我们使用了函数声明, 上述2个定义, 解释器在解释的时候都会报错, 但是可能有多种原因。

如果在全局代码里定义（也就是程序级别），解释器会将它看做是函数声明，因为他是以function关键字开头，第一个例子，我们会得到SyntaxError错误，是因为函数声明没有名字（我们前面提到了函数声明必须有名字）。

第二个例子，我们有一个名称为foo的一个函数声明正常创建，但是我们依然得到了一个语法错误——没有任何表达式的分组操作符错误。在函数声明后面他确实是一个分组操作符，而不是一个函数调用所使用的圆括号。所以如果我们声明如下代码：

```
// "foo" 是一个函数声明，在进入上下文的时候创建

alert(foo); // 函数

function foo(x) {
    alert(x);
}(1); // 这只是一个分组操作符，不是函数调用！

foo(10); // 这才是一个真正的函数调用，结果是10
```

上述代码是没有问题的，因为声明的时候产生了2个对象：一个函数声明，一个带有1的分组操作，上面的例子可以理解为如下代码：

```
// 函数声明
function foo(x) {
    alert(x);
}

// 一个分组操作符，包含一个表达式1
(1);

// 另外一个操作符，包含一个function表达式
(function () {});

// 这个操作符里，包含的也是一个表达式"foo"
("foo");

// 等等
```

如果我们定义一个如下代码（定义里包含一个语句），我们可能会说，定义歧义，会得到报错：

```
if (true) function foo() {alert(1)}
```

根据规范，上述代码是错误的（一个表达式语句不能以function关键字开头），但下面的例
本文档使用 [看云](#) 构建

子就没有报错，想想为什么？

我们如果来告诉解释器：我就像在函数声明之后立即调用，答案是很明确的，你得声明函数表达式function expression，而不是函数声明function declaration，并且创建表达式最简单的方式就是用分组操作符括号，里边放入的永远是表达式，所以解释器在解释的时候就不会出现歧义。在代码执行阶段这个的function就会被创建，并且立即执行，然后自动销毁（如果没有引用的话）。

```
(function foo(x) {
  alert(x);
})(1); // 这才是调用，不是分组操作符
```

上述代码就是我们所说的在用括号括住一个表达式，然后通过（1）去调用。

注意，下面一个立即执行的函数，周围的括号不是必须的，因为函数已经处在表达式的位置，解析器知道它处理的是在函数执行阶段应该被创建的FE，这样在函数创建后立即调用了函数。

```
var foo = {
  bar: function (x) {
    return x % 2 !== 0 ? 'yes' : 'no';
  }(1)
};

alert(foo.bar); // 'yes'
```

就像我们看到的，foo.bar是一个字符串而不是一个函数，这里的函数仅仅用来根据条件参数初始化这个属性——它创建后立即调用。

因此，“关于圆括号”问题完整的答案如下：当函数不在表达式的位置的时候，分组操作符圆括号是必须的——也就是手工将函数转化成FE。

如果解析器知道它处理的是FE，就没必要用圆括号。

除了大括号以外，如下形式也可以将函数转化为FE类型，例如：

```
// 注意是1,后面的声明
1, function () {
  alert('anonymous function is called');
}();
```

```
// 或者这个
!function () {
    alert('ECMAScript');
})();

// 其它手工转化的形式

...
```

但是，在这个例子中，圆括号是最简洁的方式。

顺便提一句，组表达式包围函数描述可以没有调用圆括号，也可包含调用圆括号，即，下面的两个表达式都是正确的FE。

实现扩展：函数语句

下面的代码，根据贵方任何一个function声明都不应该被执行：

```
if (true) {
    function foo() {
        alert(0);
    }
} else {
    function foo() {
        alert(1);
    }
}

foo(); // 1 or 0 ?实际在上不同环境下测试得出个结果不一样
```

这里有必要说明的是，按照标准，这种句法结构通常是不正确的，因为我们还记得，一个函数声明（FD）不能出现在代码块中（这里if和else包含代码块）。我们曾经讲过，FD仅出现在两个位置：程序级（Program level）或直接位于其它函数体中。

因为代码块仅包含语句，所以这是不正确的。可以出现在块中的函数的唯一位置是这些语句中的一个——上面已经讨论过的表达式语句。但是，按照定义它不能以大括号开始(既然它有别于代码块)或以一个函数关键字开始（既然它有别于FD）。

但是，在标准的错误处理章节中，它允许程序语法的扩展执行。这样的扩展之一就是我们见到的出现在代码块中的函数。在这个例子中，现今的所有存在的执行都不会抛出异常，都会处理它。但是它们都有自己的方式。

if-else分支语句的出现意味着一个动态的选择。即，从逻辑上来说，它应该是在代码执行阶段动态创建的函数表达式（FE）。但是，大多数执行在进入上下文阶段时简单的创建函数声明（FD），并使用最后声明的函数。即，函数foo将显示“1”，事实上else分支将永远不会执行。

但是，SpiderMonkey（和TraceMonkey）以两种方式对待这种情况：一方面它不会将函数作为声明处理（即，函数在代码执行阶段根据条件创建），但另一方面，既然没有括号包围（再次出现解析错误——“与FD有别”），他们不能被调用，所以也不是真正的函数表达式，它储存在变量对象中。

我个人认为这个例子中SpiderMonkey的行为是正确的，拆分了它自身的函数中间类型——（FE+FD）。这些函数在合适的时间创建，根据条件，也不像FE，倒像一个可以从外部调用的FD，SpiderMonkey将这种语法扩展称之为函数语句（缩写为FS）；该语法在MDC中提及过。

命名函数表达式的特性

当函数表达式FE有一个名称（称为命名函数表达式，缩写为NFE）时，将会出现一个重要的特点。从定义（正如我们从上面示例中看到的那样）中我们知道函数表达式不会影响一个上下文的变量对象（那样意味着既不可能通过名称在函数声明之前调用它，也不可能是在声明之后调用它）。但是，FE在递归调用中可以通过名称调用自身。

```
(function foo(bar) {
    if (bar) {
        return;
    }

    foo(true); // "foo" 是可用的
})();

// 在外部，是不可用的
foo(); // "foo" 未定义
```

“foo”储存在什么地方？在foo的活动对象中？不是，因为在foo中没有定义任何“foo”。在上下文的父变量对象中创建foo？也不是，因为按照定义——FE不会影响VO(变量对象)——从外部调用foo我们可以实实在在的看到。那么在哪里呢？

以下是关键点。当解释器在代码执行阶段遇到命名的FE时，在FE创建之前，它创建了辅助的特定对象，并添加到当前作用域链的最前端。然后它创建了FE，此时（正如我们在第四章作
本文档使用 [看云](#) 构建

用域链知道的那样) 函数获取了 `[[Scope]]` 属性——创建这个函数上下文的作用域链)。此后, FE 的名称添加到特定对象上作为唯一的属性; 这个属性的值是引用到 FE 上。最后一步是从父作用域链中移除那个特定的对象。让我们在伪码中看看这个算法:

```
specialObject = {};

Scope = specialObject + Scope;

foo = new FunctionExpression;
foo.[[Scope]] = Scope;
specialObject.foo = foo; // {DontDelete}, {ReadOnly}

delete Scope[0]; // 从作用域链中删除定义的特殊对象specialObject
```

因此, 在函数外部这个名称不可用的 (因为它不在父作用域链中), 但是, 特定对象已经存储在函数的 `[[scope]]` 中, 在那里名称是可用的。

但是需要注意的是, 一些实现 (如 Rhino) 不是在特定对象中而是在 FE 的激活对象中存储这个可选的名称。Microsoft 中的执行完全打破了 FE 规则, 它在父变量对象中保持了这个名称, 这样函数在外部变得可以访问。

NFE 与 SpiderMonkey

我们来看看 NFE 和 SpiderMonkey 的区别, SpiderMonkey 的一些版本有一个与特定对象相关的属性, 它可以作为 bug 来对待 (虽然按照标准所有的都那样实现了, 但更像一个 ECMAScript 标准上的 bug)。它与标识符的解析机制相关: 作用域链的分析是二维的, 在标识符的解析中, 同样考虑到作用域链中每个对象的原型链。

如果我们在 `Object.prototype` 中定义一个属性, 并引用一个 “不存在 (nonexistent)” 的变量。我们就能看到这种执行机制。这样, 在下面示例的 “x” 解析中, 我们将到达全局对象, 但是没发现 “x”。但是, 在 SpiderMonkey 中全局对象继承了 `Object.prototype` 中的属性, 相应地, “x” 也能被解析。

```
Object.prototype.x = 10;

(function () {
  alert(x); // 10
})();
```

活动对象没有原型。按照同样的起始条件, 在上面的例子中, 不可能看到内部函数的这种行为。如果定义一个局部变量 “x”, 并定义内部函数 (FD 或匿名的 FE), 然后再内部函数中

引用“ x ”。那么这个变量将在父函数上下文（即，应该在哪里被解析）中而不是在Object.prototype中被解析。

```
Object.prototype.x = 10;

function foo() {
    var x = 20;

    // 函数声明

    function bar() {
        alert(x);
    }

    bar(); // 20, 从foo的变量对象AO中查询

    // 匿名函数表达式也是一样

    (function () {
        alert(x); // 20, 也是从foo的变量对象AO中查询
    })();
}

foo();
```

尽管如此，一些执行会出现例外，它给活动对象设置了一个原型。因此，在Blackberry 的执行中，上面例子中的“ x ”被解析为“ 10 ”。也就是说，既然在Object.prototype中已经找到了foo的值，那么它就不会到达foo的活动对象。

```
A0(bar FD or anonymous FE) -> no ->
A0(bar FD or anonymous FE).[[Prototype]] -> yes - 10
```

在SpiderMonkey 中，同样的情形我们完全可以在命名FE的特定对象中看到。这个特定的对象（按照标准）是普通对象——“就像表达式new Object()”，相应地，它应该从Object.prototype 继承属性，这恰恰是我们在SpiderMonkey（1.7以上的版本）看到的执行。其余的执行（包括新的TraceMonkey）不会为特定的对象设置一个原型。

```
function foo() {
    var x = 10;

    (function bar() {
        alert(x); // 20, 不上10, 不是从foo的活动对象上得到的
```

```

    // "x"从链上查找:
    // A0(bar) - no -> __specialObject(bar) -> no
    // __specialObject(bar).[[Prototype]] - yes: 20

    })();
}

Object.prototype.x = 20;

foo();

```

NFE与Jscript

当前IE浏览器(直到JScript 5.8 — IE8)中内置的JScript 执行有很多与函数表达式 (NFE) 相关的bug。所有的这些bug都完全与ECMA-262-3标准矛盾；有些可能会导致严重的错误。

首先，这个例子中JScript 破坏了FE的主要规则，它不应该通过函数名存储在变量对象中。可选的FE名称应该存储在特定的对象中，并只能在函数自身(而不是别的地方)中访问。但IE直接将它存储在父变量对象中。此外，命名的FE在JScript 中作为函数声明 (FD) 对待。即创建于进入上下文的阶段，在源代码中的定义之前可以访问。

```

// FE 在变量对象里可见
testNFE();

(function testNFE() {
    alert('testNFE');
});

// FE 在定义结束以后也可见
// 就像函数声明一样
testNFE();

```

正如我们所见，它完全违背了规则。

其次，在声明中将命名FE赋给一个变量时，JScript 创建了两个不同的函数对象。逻辑上（特别注意的是在NFE的外部它的名称根本不应该被访问）很难命名这种行为。

```

var foo = function bar() {
    alert('foo');
};

alert(typeof bar); // "function",

// 有趣的是
alert(foo === bar); // false!

```

```
foo.x = 10;
alert(bar.x); // 未定义

// 但执行的时候结果一样

foo(); // "foo"
bar(); // "foo"
```

再次看到，已经乱成一片了。

但是，需要注意的是，如果与变量赋值分开，单独描述NFE（如通过组运算符），然后将它赋给一个变量，并检查其相等性，结果为true，就好像是一个对象。

```
(function bar() {});

var foo = bar;

alert(foo === bar); // true

foo.x = 10;
alert(bar.x); // 10
```

此时是可以解释的。实际上，再次创建两个对象，但那样做事实上仍保持一个。如果我们再次认为这里的NFE被作为FD对待，然后在进入上下文阶段创建FD bar。此后，在代码执行阶段第二个对象——函数表达式（FE）bar被创建，它不会被存储。相应地，没有FE bar的任何引用，它被移除了。这样就只有一个对象——FD bar，对它的引用赋给了变量foo。

第三，就通过arguments.callee间接引用一个函数而言，它引用的是被激活的那个对象的名称（确切的说——再这里有两个函数对象。

```
var foo = function bar() {
    alert([
        arguments.callee === foo,
        arguments.callee === bar
    ]);
};

foo(); // [true, false]
bar(); // [false, true]
```

第四，JScript 像对待普通的FD一样对待NFE，他不服从条件表达式规则。即，就像一个FD,NFE在进入上下文时创建，在代码中最后的定义被使用。


```

var foo = function bar() {
    alert(1);
};

if (false) {

    foo = function bar() {
        alert(2);
    };

}
bar(); // 2
foo(); // 1

```

这种行为从“逻辑上”也可以解释。在进入上下文阶段，最后遇到的FD bar被创建，即包含alert(2)的函数。此后，在代码执行阶段，新的函数——FE bar创建，对它的引用赋给了变量foo。这样foo激活产生alert(1)。逻辑很清楚，但考虑到IE的bug，既然执行明显被破坏，并依赖于JScript的bug，我给单词“逻辑上(logically)”加上了引号。

JScript的第五个bug与全局对象的属性创建相关，全局对象由赋值给一个未限定的标识符（即，没有var关键字）来生成。既然NFE在这被作为FD对待，相应地，它存储在变量对象中，赋给一个未限定的标识符（即不是赋给变量而是全局对象的普通属性），万一函数的名称与未限定的标识符相同，这样该属性就不是全局的了。

```

(function () {

    // 不用var的话，就不是当前上下文的一个变量了
    // 而是全局对象的一个属性

    foo = function foo() {};

})();

// 但，在匿名函数的外部，foo这个名字是不可用的

alert(typeof foo); // 未定义

```

“逻辑”已经清楚了：在进入上下文阶段，函数声明foo取得了匿名函数局部上下文的活动对象。在代码执行阶段，名称foo在AO中已经存在，即，它被作为局部变量。相应地，在赋值操作中，只是简单的更新已存在于AO中的属性foo，而不是按照ECMA-262-3的逻辑创建全局对象的新属性。

通过函数构造器创建的函数

既然这种函数对象也有自己的特色，我们将它与FD和FE区分开来。其主要特点在于这种函数的[[Scope]]属性仅包含全局对象：

```
var x = 10;

function foo() {
    var x = 20;
    var y = 30;

    var bar = new Function('alert(x); alert(y);');

    bar(); // 10, "y" 未定义
}
```

我们看到，函数bar的[[Scope]]属性不包含foo上下文的Ao——变量“y”不能访问，变量“x”从全局对象中取得。顺便提醒一句，Function构造器既可使用new关键字，也可以没有，这样说来，这些变体是等价的。

这些函数的其他特点与[Equated Grammar Productions](#)和[Joined Objects](#)相关。作为优化建议（但是，实现上可以不使用优化），规范提供了这些机制。如，如果我们有一个100个元素的数组，在函数的一个循环中，执行可能使用Joined Objects机制。结果是数组中的所有元素仅一个函数对象可以使用。

```
var a = [];

for (var k = 0; k < 100; k++) {
    a[k] = function () {}; // 可能使用了joined objects
}
```

但是通过函数构造器创建的函数不会被连接。

```
var a = [];

for (var k = 0; k < 100; k++) {
    a[k] = Function(''); // 一直是100个不同的函数
}
```

另外一个与联合对象（joined objects）相关的例子：

```
function foo() {
```

```

    function bar(z) {
        return z * z;
    }

    return bar;
}

var x = foo();
var y = foo();

```

这里的实现，也有权利连接对象x和对象y（使用同一个对象），因为函数（包括它们的内部[[Scope]]属性）在根本上是没有区别的。因此，通过函数构造器创建的函数总是需要更多的内存资源。

创建函数的算法

下面的伪码描述了函数创建的算法（与联合对象相关的步骤除外）。这些描述有助于你理解ECMAScript中函数对象的更多细节。这种算法适合所有的函数类型。

```

F = new NativeObject();

// 属性[[Class]]是"Function"
F.[[Class]] = "Function"

// 函数对象的原型是Function的原型
F.[[Prototype]] = Function.prototype

// 医用到函数自身
// 调用表达式F的时候激活[[Call]]
// 并且创建新的执行上下文
F.[[Call]] = <reference to function>

// 在对象的普通构造器里编译
// [[Construct]] 通过new关键字激活
// 并且给新对象分配内存
// 然后调用F.[[Call]]初始化作为this传递的新创建的对象
F.[[Construct]] = internalConstructor

// 当前执行上下文的作用域链
// 例如，创建F的上下文
F.[[Scope]] = activeContext.Scope
// 如果函数通过new Function(...)来创建，
// 那么
F.[[Scope]] = globalContext.Scope

// 传入参数的个数
F.length = countParameters

// F对象创建的原型
__objectPrototype = new Object();

```

```
__objectPrototype.constructor = F // {DontEnum}, 在循环里不可枚举x
F.prototype = __objectPrototype

return F
```

注意，F.[[Prototype]]是函数（构造器）的一个原型，F.prototype是通过这个函数创建的对象的原型（因为术语常常混乱，一些文章中F.prototype被称之为“构造器的原型”，这是不正确的）。

结论

这篇文章有些长。但是，当我们在接下来关于对象和原型章节中将继续讨论函数，同样，我很乐意在评论中回答您的任何问题。

其它参考

- 13. — [Function Definition](#);
- 15.3 — [Function Objects](#).

(16) 闭包 (Closures)

介绍

本章我们将介绍在JavaScript里大家经常来讨论的话题 —— 闭包 (closure)。闭包其实大家都已经谈烂了。尽管如此，这里还是要试着从理论角度来讨论下闭包，看看ECMAScript中的闭包内部究竟是如何工作的。

正如在前面的文章中提到的，这些文章都是系列文章，相互之间都是有关联的。因此，为了更好的理解本文要介绍的内容，建议先去阅读第[14章作用域链](#)和第[12章变量对象](#)。

英文原文：<http://dmitrysoshnikov.com/ecmascript/chapter-6-closures/>

概论

在直接讨论ECMAScript闭包之前，还是有必要来看一下函数式编程中一些基本定义。

众所周知，在函数式语言中（ECMAScript也支持这种风格），函数即是数据。就比方说，函数可以赋值给变量，可以当参数传递给其他函数，还可以从函数里返回等等。这类函数有特殊的名字和结构。

定义

A functional argument (“Funarg”) — is an argument which value is a function.
函数式参数 (“Funarg”) —— 是指值为函数的参数。

例子：

```
function exampleFunc(funArg) {  
    funArg();  
}  
  
exampleFunc(function () {  
    alert('funArg');  
});
```

上述例子中funarg的实际参数其实是传递给exampleFunc的匿名函数。

反过来，接受函数式参数的函数称为高阶函数 (high-order function 简称：HOF)。还可以称作：函数式函数或者偏数理或操作符。上述例子中，exampleFunc 就是这样的函数。

此前提到的，函数不仅可以作为参数，还可以作为返回值。这类以函数为返回值的函数称为带函数值的函数 (functions with functional value or function valued functions) 。

```
(function functionValued() {
  return function () {
    alert('returned function is called');
  };
})();
```

可以以正常数据形式存在的函数 (比方说：当参数传递，接受函数式参数或者以函数值返回) 都称作 第一类函数 (一般说第一类对象)。在ECMAScript中，所有的函数都是第一类对象。

函数可以作为正常数据存在 (例如：当参数传递，接受函数式参数或者以函数值返回) 都称作第一类函数 (一般说第一类对象)。

在ECMAScript中，所有的函数都是第一类对象。

接受自己作为参数的函数，称为自应用函数 (auto-applicative function 或者 self-applicative function)：

```
(function selfApplicative(funArg) {
  if (funArg && funArg === selfApplicative) {
    alert('self-applicative');
    return;
  }
  selfApplicative(selfApplicative);
})();
```

以自己为返回值的函数称为自复制函数 (auto-replicative function 或者 self-replicative function)。通常，“自复制”这个词用在文学作品中：

```
(function selfReplicative() {
  return selfReplicative;
})();
```

自复制函数的其中一个比较有意思的模式是让仅接受集合的一个项作为参数来接受从而代替接受集合本身。

```

// 接受集合的函数
function registerModes(modes) {
  modes.forEach(registerMode, modes);
}

// 用法
registerModes(['roster', 'accounts', 'groups']);

// 自复制函数的声明
function modes(mode) {
  registerMode(mode); // 注册一个mode
  return modes; // 返回函数自身
}

// 用法, modes链式调用
modes('roster')('accounts')('groups')

//有点类似: jQueryObject.addClass("a").toggle().removeClass("b")

```

但直接传集合用起来相对来说，比较有效并且直观。

在函数式参数中定义的变量，在“funarg”激活时就能够访问了（因为存储上下文数据的变量对象每次在进入上下文的时候就创建出来了）：

```

function testFn(funArg) {
  // funarg激活时，局部变量localVar可以访问了
  funArg(10); // 20
  funArg(20); // 30
}

testFn(function (arg) {
  var localVar = 10;
  alert(arg + localVar);
});

```

然而，我们从第14章知道，在ECMAScript中，函数是可以封装在父函数中的，并可以使用父函数上下文的变量。这个特性会引发funarg问题。

Funarg问题

在[面向堆栈的编程语言](#)中，函数的局部变量都是保存在栈上的，每当函数激活的时候，这些变量和函数参数都会压入到该堆栈上。

当函数返回的时候，这些参数又会从栈中移除。这种模型对将函数作为函数式值使用的时候有很大的限制（比方说，作为返回值从父函数中返回）。绝大部分情况下，问题会出现在当函数有自由变量的时候。

自由变量是指在函数中使用的，但既不是函数参数也不是函数的局部变量的变量

例子：

```
function testFn() {
    var localVar = 10;

    function innerFn(innerParam) {
        alert(innerParam + localVar);
    }

    return innerFn;
}

var someFn = testFn();
someFn(20); // 30
```

上述例子中，对于innerFn函数来说，localVar就属于自由变量。

对于采用面向栈模型来存储局部变量的系统而言，就意味着当testFn函数调用结束后，其局部变量都会从堆栈中移除。这样一来，当从外部对innerFn进行函数调用的时候，就会发生错误（因为localVar变量已经不存在了）。

而且，上述例子在面向栈实现模型中，要想将innerFn以返回值返回根本是不可能的。因为它也是testFn函数的局部变量，也会随着testFn的返回而移除。

还有一个问题是当系统采用动态作用域，函数作为函数参数使用的时候有关。

看如下例子（伪代码）：

```
var z = 10;

function foo() {
    alert(z);
}

foo(); // 10 - 使用静态和动态作用域的时候

(function () {
    var z = 20;
    foo(); // 10 - 使用静态作用域, 20 - 使用动态作用域
})();

// 将foo作为参数的时候是一样的
(function (funArg) {
```

本文档使用 [看云](#) 构建


```

var z = 30;
funArg(); // 10 - 静态作用域, 30 - 动态作用域

})(foo);

```

我们看到，采用动态作用域，变量（标识符）的系统是通过变量动态栈来管理的。因此，自由变量是在当前活跃的动态链中查询的，而不是在函数创建的时候保存起来的静态作用域链中查询的。

这样就会产生冲突。比方说，即使Z仍然存在（与之前从栈中移除变量的例子相反），还是会有这样一个问题：在不同的函数调用中，Z的值到底取哪个呢（从哪个上下文，哪个作用域中查询）？

上述描述的就是两类funarg问题——取决于是否将函数以返回值返回（第一类问题）以及是否将函数当函数参数使用（第二类问题）。

为了解决上述问题，就引入了闭包的概念。

闭包

闭包是代码块和创建该代码块的上下文中数据的结合。

让我们来看下面这个例子（伪代码）：

```

var x = 20;

function foo() {
  alert(x); // 自由变量"x" == 20
}

// 为foo闭包
fooClosure = {
  call: foo // 引用到function
  lexicalEnvironment: {x: 20} // 搜索上下文的上下文
};

```

上述例子中，“fooClosure”部分是伪代码。对应的，在ECMAScript中，“foo”函数已经有了一个内部属性——创建该函数上下文的作用域链。

“lexical”通常是省略的。上述例子中是为了强调在闭包创建的同时，上下文的数据就会保存起来。当下次调用该函数的时候，自由变量就可以在保存的（闭包）上下文中找到了，正如上述代码所示，变量“z”的值总是10。

定义中我们使用的比较广义的词 —— “代码块”，然而，通常（在ECMAScript中）会使用我们经常用到的函数。当然了，并不是所有对闭包的实现都会将闭包和函数绑在一起，比方说，在Ruby语言中，闭包就有可能是：一个过程对象（procedure object），一个lambda表达式或者是代码块。

对于要实现将局部变量在上下文销毁后仍然保存下来，基于栈的实现显然是不适用的（因为与基于栈的结构相矛盾）。因此在这种情况下，上层作用域的闭包数据是通过动态分配内存的方式来实现的（基于“堆”的实现），配合使用垃圾回收器（garbage collector简称GC）和引用计数（reference counting）。这种实现方式比基于栈的实现性能要低，然而，任何一种实现总是可以优化的：可以分析函数是否使用了自由变量，函数式参数或者函数式值，然后根据情况来决定 —— 是将数据存放在堆栈中还是堆中。

ECMAScript闭包的实现

讨论完理论部分，接下来让我们来介绍下ECMAScript中闭包究竟是如何实现的。这里还是有必要再次强调下：ECMAScript只使用[静态（词法）作用域](#)（而诸如Perl这样的语言，既可以使用静态作用域也可以使用动态作用域进行变量声明）。

```
var x = 10;

function foo() {
    alert(x);
}

(function (funArg) {

    var x = 20;

    // 变量"x"在(lexical)上下文中静态保存的，在该函数创建的时候就保存了
    funArg(); // 10，而不是20

})(foo);
```

技术上说，创建该函数的父级上下文的数据是保存在函数的内部属性 `[[Scope]]` 中的。如果你还不了解什么是 `[[Scope]]`，建议你先阅读第14章，该章节对 `[[Scope]]` 作了非常详细的介绍。如果你对 `[[Scope]]` 和作用域链的知识完全理解了的话，那对闭包也就完全理解了。

根据函数创建的算法，我们看到在ECMAScript中，所有的函数都是闭包，因为它们都是在创建的时候就保存了上层上下文的作用域链（除开异常的情况）（不管这个函数后续是否会激活 —— `[[Scope]]` 在函数创建的时候就有了）：

```
var x = 10;
```

```
function foo() {
    alert(x);
}

// foo是闭包
foo: = {
    [[Call]]: ,
    [[Scope]]: [
        global: {
            x: 10
        }
    ],
    ... // 其它属性
};
```

如我们所说，为了优化目的，当一个函数没有使用自由变量的话，实现可能不保存在副作用域链里。不过，在ECMA-262-3规范里任何都没说。因此，正常来说，所有的参数都是在创建阶段保存在[[Scope]]属性里的。

有些实现中，允许对闭包作用域直接进行访问。比如Rhino，针对函数的[[Scope]]属性，对应有一个非标准的 parent属性，在第12章中作过介绍：

```
var global = this;
var x = 10;

var foo = (function () {

    var y = 20;

    return function () {
        alert(y);
    };

})();

foo(); // 20
alert(foo.__parent__.y); // 20

foo.__parent__.y = 30;
foo(); // 30

// 可以通过作用域链移动到顶部
alert(foo.__parent__.__parent__ === global); // true
alert(foo.__parent__.__parent__.x); // 10
```

所有对象都引用一个[[Scope]]

这里还要注意的：在ECMAScript中，同一个父上下文中创建的闭包是共用一个[[Scope]]

属性的。也就是说，某个闭包对其中[[Scope]]的变量做修改会影响到其他闭包对其变量的读取：

这就是说：所有的内部函数都共享同一个父作用域

```
var firstClosure;
var secondClosure;

function foo() {

    var x = 1;

    firstClosure = function () { return ++x; };
    secondClosure = function () { return --x; };

    x = 2; // 影响 A0["x"], 在2个闭包公有的[[Scope]]中

    alert(firstClosure()); // 3, 通过第一个闭包的[[Scope]]
}

foo();

alert(firstClosure()); // 4
alert(secondClosure()); // 3
```

关于这个功能有一个非常普遍的错误认识，开发人员在循环语句里创建函数（内部进行计数）的时候经常得不到预期的结果，而期望是每个函数都有自己的值。

```
var data = [];

for (var k = 0; k < 3; k++) {
    data[k] = function () {
        alert(k);
    };
}

data[0](); // 3, 而不是0
data[1](); // 3, 而不是1
data[2](); // 3, 而不是2
```

上述例子就证明了——同一个上下文中创建的闭包是共用一个[[Scope]]属性的。因此上层上下文中的变量“k”是可以很容易就被改变的。

```
activeContext.Scope = [
    ... // 其它变量对象
    {data: [...], k: 3} // 活动对象
];
```

```
data[0].[[Scope]] === Scope;
data[1].[[Scope]] === Scope;
data[2].[[Scope]] === Scope;
```

这样一来，在函数激活的时候，最终使用到的k就已经变成了3了。如下所示，创建一个闭包就可以解决这个问题了：

```
var data = [];

for (var k = 0; k < 3; k++) {
  data[k] = (function _helper(x) {
    return function () {
      alert(x);
    };
  })(k); // 传入"k"值
}

// 现在结果是正确的了
data[0](); // 0
data[1](); // 1
data[2](); // 2
```

让我们来看看上述代码都发生了什么？函数 “_helper” 创建出来之后，通过传入参数 “k” 激活。其返回值也是个函数，该函数保存在对应的数组元素中。这种技术产生了如下效果：在函数激活时，每次 “_helper” 都会创建一个新的变量对象，其中含有参数 “x”，“x” 的值就是传递进来的 “k” 的值。这样一来，返回的函数的[[Scope]]就成了如下所示：

```
data[0].[[Scope]] === [
  ... // 其它变量对象
  父级上下文中的活动对象A0: {data: [...], k: 3},
  _helper上下文中的活动对象A0: {x: 0}
];

data[1].[[Scope]] === [
  ... // 其它变量对象
  父级上下文中的活动对象A0: {data: [...], k: 3},
  _helper上下文中的活动对象A0: {x: 1}
];

data[2].[[Scope]] === [
  ... // 其它变量对象
  父级上下文中的活动对象A0: {data: [...], k: 3},
  _helper上下文中的活动对象A0: {x: 2}
];
```

我们看到，这时函数的[[Scope]]属性就有了真正想要的值了，为了达到这样的目的，我们不得不在[[Scope]]中创建额外的变量对象。要注意的是，在返回的函数中，如果要获取“k”的值，那么该值还是会3。

顺便提下，大量介绍JavaScript的文章都认为只有额外创建的函数才是闭包，这种说法是错误的。实践得出，这种方式是最有效的，然而，从理论角度来说，在ECMAScript中所有的函数都是闭包。

然而，上述提到的方法并不是唯一的方法。通过其他方式也可以获得正确的“k”的值，如下所示：

```
var data = [];

for (var k = 0; k < 3; k++) {
  (data[k] = function () {
    alert(arguments.callee.x);
  }).x = k; // 将k作为函数的一个属性
}

// 结果也是对的
data[0](); // 0
data[1](); // 1
data[2](); // 2
```

Funarg和return

另外一个特性是从闭包中返回。在ECMAScript中，闭包中的返回语句会将控制流返回给调用上下文（调用者）。而在其他语言中，比如，Ruby，有很多中形式的闭包，相应的处理闭包返回也都不同，下面几种方式都是可能的：可能直接返回给调用者，或者在某些情况下——直接从上下文退出。

ECMAScript标准的退出行为如下：

```
function getElement() {
  [1, 2, 3].forEach(function (element) {
    if (element % 2 == 0) {
      // 返回给函数"forEach"函数
      // 而不是返回给getElement函数
      alert('found: ' + element); // found: 2
      return element;
    }
  });
}
```

```
    return null;
}
```

然而，在ECMAScript中通过try catch可以实现如下效果：

```
var $break = {};

function getElement() {
    try {
        [1, 2, 3].forEach(function (element) {
            if (element % 2 == 0) {
                // // 从getElement中"返回"
                alert('found: ' + element); // found: 2
                $break.data = element;
                throw $break;
            }
        });
    } catch (e) {
        if (e == $break) {
            return $break.data;
        }
    }
    return null;
}

alert(getElement()); // 2
```

理论版本

这里说明一下，开发人员经常错误将闭包简化理解成从父上下文中返回内部函数，甚至理解成只有匿名函数才能是闭包。

再说一下，因为作用域链，使得所有的函数都是闭包（与函数类型无关：匿名函数，FE，NFE，FD都是闭包）。

这里只有一类函数除外，那就是通过Function构造器创建的函数，因为其[[Scope]]只包含全局对象。

为了更好的澄清该问题，我们对ECMAScript中的闭包给出2个正确的版本定义：

ECMAScript中，闭包指的是：

1. 从理论角度：所有的函数。因为它们都在创建的时候就将上层上下文的数据保存起来了。哪怕是简单的全局变量也是如此，因为函数中访问全局变量就相当于是在访问自由变量，这个时候使用最外层的作用域。
2. 从实践角度：以下函数才算是闭包：
 1. 即使创建它的上下文已经销毁，它仍然存在（比如，内部函数从父函数中返回）
 2. 在代码中引用了自由变量

闭包用法实战

实际使用的时候，闭包可以创建出非常优雅的设计，允许对funarg上定义的多种计算方式进行定制。如下就是数组排序的例子，它接受一个排序条件函数作为参数：

```
[1, 2, 3].sort(function (a, b) {
  ... // 排序条件
});
```

同样的例子还有，数组的map方法是根据函数中定义的条件将原数组映射到一个新的数组中：

```
[1, 2, 3].map(function (element) {
  return element * 2;
}); // [2, 4, 6]
```

使用函数式参数，可以很方便的实现一个搜索方法，并且可以支持无限制的搜索条件：

```
someCollection.find(function (element) {
  return element.someProperty == 'searchCondition';
});
```

还有应用函数，比如常见的forEach方法，将函数应用到每个数组元素：

```
[1, 2, 3].forEach(function (element) {
  if (element % 2 != 0) {
    alert(element);
  }
}); // 1, 3
```

顺便提下，函数对象的 apply 和 call 方法，在函数式编程中也可以用作应用函数。apply 和 call 已经在讨论 “this” 的时候介绍过了；这里，我们将它们看作是应用函数——应用到参数中的函数（在 apply 中是参数列表，在 call 中是独立的参数）：


```
(function () {  
    alert([].join.call(arguments, ';')); // 1;2;3  
}).apply(this, [1, 2, 3]);
```

闭包还有另外一个非常重要的应用 —— 延迟调用：

```
var a = 10;  
setTimeout(function () {  
    alert(a); // 10, after one second  
, 1000);
```

还有回调函数

```
//...  
var x = 10;  
// only for example  
xmlHttpRequestObject.onreadystatechange = function () {  
    // 当数据就绪的时候，才会调用；  
    // 这里，不论是在哪个上下文中创建  
    // 此时变量“x”的值已经存在了  
    alert(x); // 10  
};  
//...
```

还可以创建封装的作用域来隐藏辅助对象：

```
var foo = {};  
  
// 初始化  
(function (object) {  
  
    var x = 10;  
  
    object.getX = function _getX() {  
        return x;  
    };  
  
})(foo);  
  
alert(foo.getX()); // 获得闭包 "x" - 10
```

总结

本文介绍了更多关于ECMAScript-262-3的理论知识，而我认为，这些基础的理论有助于理解ECMAScript中闭包的概念。如果有任何问题，我回在评论里回复大家。

其它参考

- [Javascript Closures \(by Richard Cornford\)](#)
- [Funarg problem](#)
- [Closures](#)

(17) 面向对象编程之一般理论

介绍

在本篇文章，我们考虑在ECMAScript中的面向对象编程的各个方面（虽然以前在许多文章中已经讨论过这个话题）。我们将更多地从理论方面看这些问题。特别是，我们会考虑对象的创建算法，对象（包括基本关系 - 继承）之间的关系是如何，也可以在讨论中使用（我希望将消除之前对于JavaScript中OOP的一些概念歧义）。

英文原文：<http://dmitrysoshnikov.com/ecmascript/chapter-7-1-oop-general-theory/>

概论、范式与思想

在进行ECMAScript中的OOP技术分析之前，我们有必要掌握一些OOP基本的特征，并澄清概论中的主要概念。

ECMAScript支持包括结构化、面向对象、函数式、命令式等多种编程方式，某些情况下还支持面向方面编程；但本文是讨论面向对象编程，所以来给出ECMAScript中面向对象编程的定义：

ECMAScript是基于原型实现的面向对象编程语言。

基于原型的OOP和基于静态类的方式直接有很多差异。让我们一起来看看他们直接详细的差异。

基于类特性和基于原型

注意，在前面一句很重要的一点已经指出的那样-完全基于静态类。随着“静态”一词，我们了解静态对象和静态类，强类型（虽然不是必需的）。

关于这种情况，很多论坛上的文档都有强调这是他们反对将在JavaScript里将“类与原型”进行比较的主要原因，尽管他们在实现上的有所不同（例如基于动态类的Python和Ruby）不是太反对的重点（某些条件写，尽管思想上有一定不同，但JavaScript没有变得那么另类），但他们反对的重点是静态类和动态原型(statics + classes vs. dynamics + prototypes)，确切地说，一个静态类（例如：C + +，JAVA）和他的属下及方法定义的机制可以让我们看到它和基于原型实现的准确区别。

但是，让我们来一个一个列举一下。让我们考虑总则和这些范式的主要概念。

基于静态类

在基于类的模型中，有个关于类和实例的概念。类的实例也常常被命名为对象或范例。

类与对象

类代表了一个实例（也就是对象）的抽象。在这方面有点像数学，但我们一把称之为类型（type）或分类（classification）。

例如（这里和下面的例子都是伪代码）：

```
C = Class {a, b, c} // 类C，包括特性a, b, c
```

实例的特点是：属性（对象描述）和方法（对象活动）。特性本身也可视为对象：即属性是否可写的，可配置，可设置的（getter/setter）等。因此，对象存储了状态（即在一个类中描述的所有属性的具体值），类为他们的实例定义了严格不变的结构（属性）和严格不变的行为（方法）。

```
C = Class {a, b, c, method1, method2}

c1 = {a: 10, b: 20, c: 30} // 类C是实例：对象c1
c2 = {a: 50, b: 60, c: 70} // 类C是实例：对象c2，拥有自己的状态（也就是属性值）
```

层次继承

为了提高代码重用，类可以从一个扩展为另一个，在加上额外的信息。这种机制被称为（分层）继承。

```
D = Class extends C = {d, e} // {a, b, c, d, e}
d1 = {a: 10, b: 20, c: 30, d: 40, e: 50}
```

在类的实例上调用方的时候，通常会现在原生类本书就查找该方法，如果没找到就到直接父类去查找，如果还没找到，就到父类的父类去查找（例如严格的继承链上），如果查到继承的顶部还没查到，那结果就是：该对象没有类似的行为，也没办法获取结果。

```
d1.method1() // D.method1 (no) -> C.method1 (yes)
d1.method5() // D.method5 (no) -> C.method5 (no) -> no result
```

与在继承里方法不复制到一个子类相比，属性总是被复杂到子类里的。我们可以看到子类D继承自父类C类：属性a,b,c是复制过去了，D的结构是{a, b, c, d, e}。然而，方法

{method1, method2}是没有复制过去，而是继承过去的。因此，也就是说如果一个很深层次的类有一些对象根本不需要的属性的话，那子类也有拥有这些属性。

基于类的关键概念

因此，我们有如下关键概念：

1. 创建一个对象之前，必须声明类，首先有必要界定其类
2. 因此，该对象将由抽象成自身“象形和相似性”（结构和行为）的类里创建
3. 方法是通过了严格的，直接的，一成不变的继承链来处理
4. 子类包含了继承链中所有的属性（即使其中的某些属性是子类不需要的）；
5. 创建类实例，类不能（因为静态模型）来改变其实例的特征（属性或方法）；
6. 实例（因为严格的静态模型）除了有该实例所对应类里声明的行为和属性以外，是不能额外的行为或属性的。

让我们看看在JavaScript里如何替代OOP模型，也就是我们所建议的基于原型的OOP。

基于原型

这里的基本概念是动态可变对象。转换（完整转换，不仅包括值，还包括特性）和动态语言有直接关系。下面这样的对象可以独立存储他们所有的特性（属性，方法）而不需要的类。

```
object = {a: 10, b: 20, c: 30, method: fn};
object.a; // 10
object.c; // 30
object.method();
```

此外，由于动态的，他们可以很容易地改变（添加，删除，修改）自己的特性：

```
object.method5 = function () {...}; // 添加新方法
object.d = 40; // 添加新属性 "d"
delete object.c; // 删除属性 "c"
object.a = 100; // 修改属性 "a"

// 结果是：object: {a: 100, b: 20, d: 40, method: fn, method5: fn};
```

也就是说，在赋值的时候，如果某些特性不存在，则创建它并且将赋值与它进行初始化，如果它存在，就只是更新。

在这种情况下，代码重用不是通过扩展类来实现的，（请注意，我们没有说类没办法改变，因为这里根本没有类的概念），而是通过原型来实现的。

原型是一个对象，它是用来作为其他对象的原始copy，或者如果一些对象没有自己的必要特性，原型可以作为这些对象的一个委托而当成辅助对象。

基于委托

任何对象都可以被用来作为另一个对象的原型对象，因为对象可以很容易地在运行时改变它的原型动态。

注意，目前我们正在考虑的是概论而不是具体实现，当我们在ECMAScript中讨论具体实现时，我们将看到他们自身的一些特点。

例（伪代码）：

```
x = {a: 10, b: 20};
y = {a: 40, c: 50};
y.[[Prototype]] = x; // x是y的原型

y.a; // 40, 自身特性
y.c; // 50, 也是自身特性
y.b; // 20 - 从原型中获取: y.b (no) -> y.[[Prototype]].b (yes): 20

delete y.a; // 删除自身的"a"
y.a; // 10 - 从原型中获取

z = {a: 100, e: 50}
y.[[Prototype]] = z; // 将y的原型修改为z
y.a; // 100 - 从原型z中获取
y.e // 50, 也是从原型z中获取

z.q = 200 // 添加新属性到原型上
y.q // 修改也适用于y
```

这个例子展示了原型作为辅助对象属性的重要功能和机制，就像是要自己的属性一下，和自身属性相比，这些属性是委托属性。这个机制被称为委托，并且基于它的原型模型是一个委托的原型（或基于委托的原型）。引用的机制在这里称为发送信息到对象上，如果这个对象得不到响应就会委托给原型来查找（要求它尝试响应消息）。

在这种情况下的代码重用被称为基于委托的继承或基于原型的继承。由于任何对象可以当成原型，也就是说原型也可以有自己的原型。这些原型连接在一起形成一个所谓的原型链。链也像静态类中分层次的，但是它可以很容易地重新排列，改变层次和结构。

```
x = {a: 10}

y = {b: 20}
y.[[Prototype]] = x
```

```

z = {c: 30}
z.[[Prototype]] = y

z.a // 10

// z.a 在原型链里查到:
// z.a (no) ->
// z.[[Prototype]].a (no) ->
// z.[[Prototype]].[[Prototype]].a (yes): 10

```

如果一个对象和它的原型链不能响应消息发送，该对象可以激活相应的系统信号，可能是由原型链上其它的委托进行处理。

该系统信号，在许多实现里都是可用的，包括基于活动态类的系统：Smalltalk中的#doesNotUnderstand，Ruby中的method_missing；Python中的getattr，PHP中的call；和ECMAScript中的noSuchMethod__实现，等等。

例（SpiderMonkey的ECMAScript的实现）：

```

var object = {
  // catch住不能响应消息的系统信号
  __noSuchMethod__: function (name, args) {
    alert([name, args]);
    if (name == 'test') {
      return '.test() method is handled';
    }
    return delegate[name].apply(this, args);
  }
};

var delegate = {
  square: function (a) {
    return a * a;
  }
};

alert(object.square(10)); // 100
alert(object.test()); // .test() method is handled

```

也就是说，基于静态类的实现，在不能响应消息的情况下，得出的结论是：目前的对象不具有所要求的特性，但是如果尝试从原型链里获取，依然可能得到结果，或者该对象经过一系列变化以后拥有该特性。

关于ECMAScript，具体的实现就是：使用基于委托的原型。然而，正如我们将从规范和实

现里看到的，他们也有自身的特性。

Concatenative模型

老实说，有必要在说句话关于另外一种情况（尽快在ECMAScript没有用到）：当原型从其它对象复杂原来代替原生对象这种情况。这种情况代码重用是在对象创建阶段对一个对象的真正复制（克隆）而不是委托。这种原型被称为concatenative原型。复制对象所有原型的特性，可以进一步完全改变其属性和方法,同样作为原型可以改变自己（在基于委托的模型中，这个改变不会改变现有存在的对象行为，而是改变它的原型特性）。这种方法的优点是减少调度和委托的时间，而缺点是内存使用率搞。

Duck类型

回来动态弱类型变化的对象，与基于静态类的模型相比，检验它是否可以这些事和对象有什么类型（类）无关，而是是否能够相应消息有关（即在检查以后是否有能力做它是必须的）。

例如：

```
// 在基于静态来的模型里
if (object instanceof SomeClass) {
    // 一些行为是运行的
}

// 在动态实现里
// 对象在此时是什么类型并不重要
// 因为突变、类型、特性可以自由重复的转变。
// 重要的对象是否可以响应test消息
if (isFunction(object.test)) // ECMAScript

if object.respond_to?(:test) // Ruby

if hasattr(object, 'test'): // Python
```

这就是所谓的Dock类型。也就是说，物体在check的时候可以通过自己的特性来识别，而不是对象在层次结构中的位置或他们属于任何具体类型。

基于原型的关键概念

让我们来看一下这种方式的主要特点：

1. 基本概念是对象
2. 对象是完全动态可变的（理论上完全可以从一个类型转化到另一个类型）
3. 对象没有描述自己的结构和行为的严格类，对象不需要类
4. 对象没有类但可以有原型，他们如果不能响应消息的话可以委托给原型

5. 在运行时随时可以改变对象的原型;
6. 在基于委托的模型中, 改变原型的特点, 将影响到与该原型相关的所有对象;
7. 在concatenative原型模型中, 原型是从其他对象克隆的原始副本, 并进一步成为完全独立的副本原件, 原型特性的变换不会影响从它克隆的对象
8. 如果不能响应消息, 它的调用者可以采取额外的措施 (例如, 改变调度)
9. 对象的失败可以不由它们的层次和所属哪个类来决定, 而是由当前特性来决定

不过, 还有一个模型, 我们也应该考虑。

基于动态类

我们认为, 在上面例子里展示的区别 “类VS原型 ” 在这个基于动态类的模型中不是那么重要, (尤其是如果原型链是不变的, 为更准确区分, 还是有必要考虑一个静态类)。作为例子, 它也可以使用Python或Ruby (或其他类似的语言)。这些语言都使用基于动态类的范式。然而, 在某些方面, 我们是可以看到基于原型实现的某些功能。

在下面例子中, 我们可以看到仅仅是基于委托的原型, 我们可以放大一个类 (原型), 从而影响到所有与这个类相关的对象, 我们也可以在运行时动态地改变这个对象的类 (为委托提供一个新对象) 等等。

```
# Python

class A(object):

    def __init__(self, a):
        self.a = a

    def square(self):
        return self.a * self.a

a = A(10) # 创建实例
print(a.a) # 10

A.b = 20 # 为类提供一个新属性
print(a.b) # 20 - 可以在"a"实例里访问到

a.b = 30 # 创建a自身的属性
print(a.b) # 30

del a.b # 删除自身的属性
print(a.b) # 20 - 再次从类里获取 (原型)

# 就像基于原型的模型
# 可以在运行时改变对象的原型

class B(object): # 空类B
    pass
```

```

b = B() # B的实例

b.__class__ = A # 动态改变类（原型）

b.a = 10 # 创建新属性
print(b.square()) # 100 - A类的方法这时候可用

# 可以显示删除类上的引用
del A
del B

# 但对象依然有隐式的引用，并且这些方法依然可用
print(b.square()) # 100

# 但这时候不能再改变类了
# 这是实现的特性
b.__class__ = dict # error

```

Ruby中的实现也是类似的：也使用了完全动态的类（顺便说一下在当前版本的Python中，与Ruby和ECMAScript的对比，放大类（原型）不行的），我们可以彻底改变对象（或类）的特性（在类上添加方法/属性，而这些变化会影响已经存在的对象），但是，它不能的动态改变一个对象的类。

但是，这篇文章不是专门针对Python和Ruby的，因此我们不多说了，我们来继续讨论ECMAScript本身。

但在此之前，我们还得再看一下在一些OOP里有的“语法糖”，因为很多之前关于JavaScript的文章往往会文这些问题。

本节唯一需要注意的错误句子是：“JavaScript不是类，它有原型，可以代替类”。非常有必要知道并非所有基于类的实现都是完全不一样的，即便我们可能会说“JavaScript是不同的”，但也有必要考虑（除了“类”的概念）还有其他相关的特性呢。

各种OOP实现的其它特性

本节我们简要介绍一下其它特性和各种OOP实现中关于代码重用的方式，也包括ECMAScript中的OOP实现。原因是，之前出现的关于JavaScript中关于OOP的实现是有一些习惯性的思维限制，唯一主要的要求是，应该在技术上和思想上加以证明。不能说没发现和其它OOP实现里的语法糖功能，就草率认为JavaScript不是不是纯粹的OOP语言，这是不对滴。

多态

在ECMAScript中对象有几种含义的多态性。

例如，一个函数可以应用于不同的对象，就像原生对象的特性（因为这个值在进入执行上下文时确定的）：

```
function test() {
  alert([this.a, this.b]);
}

test.call({a: 10, b: 20}); // 10, 20
test.call({a: 100, b: 200}); // 100, 200

var a = 1;
var b = 2;

test(); // 1, 2
```

不过，也有例外：Date.prototype.getTime()方法，根据标准这个值总是应该有一个日期对象，否则就会抛出异常。

```
alert(Date.prototype.getTime.call(new Date())); // time
alert(Date.prototype.getTime.call(new String(''))); // TypeError
```

所谓函数定义时的参数多态性也就等价于所有数据类型，只不过接受多态性参数（例如数组的.sort排序方法和它的参数——多态的排序功能）。顺便说一下，上面的例子也可以被视为是一种参数多态性。

原型里方法可以被定义为空，所有创建的对象应重新定义（实现）该方法（即“一个接口（签名），多个实现”）。

多态性和我们上面提到的Duck类型是有关的：即对象的类型和在层次结构中的位置不是那么重要，但如果它有所有必要的特征，它可以很容易地接受（即通用接口很重要，实现则可以多种多样）。

封装

关于封装，往往会有错误的看法。本节我们讨论一下一些OOP实现里的语法糖——也就是众所周知的修饰符：在这种情况下，我们将讨论一些OOP实现便捷的“糖”-众所周知的修饰符：private,protected和public（或者称为对象的访问级别或访问修饰符）。

在这里我要提醒一下封装的主要目的：封装是一个抽象的增加，而不是选拔个直接往你的类里写入一些东西的隐藏“恶意黑客”。

这是一个很大的错误：为了隐藏使用隐藏。

访问级别（private,protected和public），为了方便编程在很多面向对象里都已经实现了（真的是非常方便的语法糖），更抽象地描述和构建系统。

这些可以在一些实现里看出（如已经提到的Python和Ruby）。一方面（在Python中），这些private _protected属性（通过下划线这个命名规范），从外部不可访问。另一方面，Python可以通过特殊的规则从外部访问（_ClassNamefield_name）。

```
class A(object):

    def __init__(self):
        self.public = 10
        self.__private = 20

    def get_private(self):
        return self.__private

# outside:

a = A() # A的实例

print(a.public) # OK, 30
print(a.get_private()) # OK, 20
print(a.__private) # 失败, 因为只能在A里可用

# 但在Python里, 可以通过特殊规则来访问

print(a._A__private) # OK, 20
```

在Ruby里：一方面有能力来定义private和protected的特性，另一方面，也有特殊的方法（例如instance_variable_get，instance_variable_set，send等）获取封装的数据。

```
class A

  def initialize
    @a = 10
  end

  def public_method
    private_method(20)
  end

  private

  def private_method(b)
    return @a + b
  end

end
```

```

a = A.new # 新实例

a.public_method # OK, 30

a.a # 失败, @a - 是私有的实例变量

# "private_method"是私有的, 只能在A类里访问

a.private_method # 错误

# 但是有特殊的元数据方法名, 可以获取到数据

a.send(:private_method, 20) # OK, 30
a.instance_variable_get(:@a) # OK, 10

```

最主要的原因是, 程序员自己想要获得的封装 (请注意, 我特别不使用“隐藏”) 的数据。如果这些数据会以某种方式不正确地更改或有任何错误, 则全部责任都是程序员, 但不是简单的“拼写错误”或“随便改变某些字段”。但如果这种情况很频繁, 那就是很不好的编程习惯和风格, 因为通常值用公共的API来和对象“交谈”。

重复一下, 封装的基本目的是一个从辅助数据的用户中抽象出来, 而不是一个防止黑客隐藏数据。更严重的, 封装不是用private修饰数据而达到软件安全的目的。

封装辅助对象 (局部), 我们用最小的代价、本地化和预测性变化来问为公共接口的行为变化提供可行性, 这也正是封装的目的。

另外setter方法的重要目的是抽象复杂的计算。例如, element.innerHTML这个setter——抽象的语句——“现在这个元素内的HTML是如下内容”, 而在innerHTML属性的setter函数将难以计算和检查。在这种情况下, 问题大多涉及到抽象, 但封装也会发生。

封装的概念不仅仅只与OOP相关。例如, 它可以是一个简单的功能, 只封装了各种计算, 使得其抽象 (没有必要让用户知道, 例如函数Math.round (...) 是如何实现的, 用户只是简单地调用它)。它是一种封装, 注意, 我没有说他是“private, protected和public”。

ECMAScript规范的当前版本, 没有定义private, protected和public修饰符。

然而, 在实践中是有可能看到有些东西被命名为“模仿JS封装”。一般该上下文的目的是 (作为一个规则, 构造函数本身) 使用。不幸的是, 经常实施这种“模仿”, 程序员可以产生伪绝对非抽象的实体设置“getter / setter方法” (我再说一遍, 它是错误的) :

```

function A() {
    var _a; // "private" a

```

```

    this.getA = function _getA() {
        return _a;
    };

    this.setA = function _setA(a) {
        _a = a;
    };
}

var a = new A();

a.setA(10);
alert(a._a); // undefined, "private"
alert(a.getA()); // 10

```

因此，每个人都明白，对于每个创建的对象，对于的getA/setA方法也创建了，这也是导致内存增加的原因（和原型定义相比）。虽然，理论上第一种情况下可以对对象进行优化。

另外，一些JavaScript的文章经常提到“私有方法”的概念，注意：ECMA-262-3标准里没有定义任何关于“私有方法”的概念。

但是，某些情况下它可以在构造函数中创建，因为JS是意识形态的语言——对象是完全可变的并且有独特的特性（在构造函数里某些条件下，有些对象可以得到额外的方法，而其他则不行）。

此外，在JavaScript里，如果还是把封装曲解成为了不让恶意黑客在某些自动写入某些值的一种理解来代替使用setter方法，那所谓的“隐藏(hidden)”和“私有(private)”其实没有很“隐藏”，有些实现可以通过调用上下文到eval函数（可以在SpiderMonkey1.7上测试）在相关的作用域链（以及相应的所有变量对象）上获取值）。

```

eval('_a = 100', a.getA); // 或者a.setA, 因为"_a"两个方法的[[Scope]]上
a.getA(); // 100

```

或者，在实现中允许直接进入活动对象（例如Rhino），通过访问该对象的相应属性可以改变内部变量的值：

```

// Rhino
var foo = (function () {
    var x = 10; // "private"
    return function () {
        print(x);
    };
})();
foo(); // 10

```

本文档使用 [看云](#) 构建

```
foo.__parent__.x = 20;
foo(); // 20
```

有时，在JavaScript里通过在变量前加下划线来达到“private”和“protected”数据的目的（但与Python相比，这里只是命名规范）：

```
var _myPrivateData = 'testString';
```

对于括号括住执行上下文是经常使用，但对于真正的辅助数据，则和对象没有直接关联，只是方便从外部的API抽象出来：

```
(function () {
    // 初始化上下文
})();
```

多重继承

多继承是代码重用改进的一个很方便的语法糖（如果我们一次能继承一个类，为什么不能一次继承10个？）。然而由于多重继承有一些不足，才导致在实现中没有流行起来。

ECMAScript不支持多继承（即只有一个对象，可以用来作为一个直接原型），虽然其祖先自编程语言有这样的能力。但在某些实现中(如SpiderMonkey)使用noSuchMethod可以用于管理调度和委托来替代原型链。

Mixins

Mixins是代码重用的一种便捷方式。Mixins已建议作为多重继承的替代品。这些独立的元素都可以与任何对象进行混合来扩展它们的功能（因此对象也可以混合多个Mixins）。

ECMA-262-3规范没有定义“Mixins”的概念，但根据Mixins定义以及ECMAScript拥有动态可变对象，所以使用Mixins简单地扩充特性是没有障碍的。

典型的例子：

```
// helper for augmentation
Object.extend = function (destination, source) {
    for (property in source) if (source.hasOwnProperty(property)) {
        destination[property] = source[property];
    }
    return destination;
};
```



```
var X = {a: 10, b: 20};
var Y = {c: 30, d: 40};

Object.extend(X, Y); // mix Y into X
alert([X.a, X.b, X.c, X.d]); 10, 20, 30, 40
```

请注意，我采取在ECMA-262-3中被提及过的引号中的这些定义（“mixin”，“mix”），在规范里并没有这样的概念，而且不是mix而是常用的通过新特性去扩展对象。（Ruby中mixins的概念是官方定义的，mixin创建了一个包含模块的一个引用来代替简单复制该模块的所有属性到另外一个模块上——事实上是：为委托创建一个额外的对象（原型））。

Traits

Traits和mixins的概念相似，但它有很多功能（根据定义，因为可以应用mixins所以不能包含状态，因为它有可能导致命名冲突）。根据ECMAScript说明Traits和mixins遵循同样的原则，所以该规范没有定义“Traits”的概念。

接口

在一些OOP中实现的接口和mixins及traits类似。然而，与mixins及traits相比，接口强制实现类必须实现其方法签名的行为。

接口完全可以被视为抽象类。不过与抽象类相比（抽象类里的方法可以只实现一部分，另外一部分依然定义为签名），继承只能是单继承基类，但可以继承多个接口，节约这个原因，可以接口（多个混合）可以看做是多继承的替代方案。

ECMA-262-3标准既没有定义“接口”的概念，也没有定义“抽象类”的概念。然而，作为模仿，它是可以由“空”的方法（或空方法中抛出异常，告诉开发人员这个方法需要被实现）的对象来实现。

对象组合

对象组合也是一个动态代码重用技术之一。对象组合不同于高灵活性的继承，它实现了一个动态可变的委托。而这，也是基于委托原型的基本。除了动态可变原型，该对象可以为委托聚合对象（创建一个组合作为结果——聚合），并进一步发送消息到对象上，委托到该委托上。这可以两个以上的委托，因为它的动态特性决定着它可以在运行时改变。

已经提到的noSuchMethod例子是这样，但也让我们展示了如何明确地使用委托：

例如：

```
var _delegate = {
```



```

    foo: function () {
        alert('_delegate.foo');
    }
};

var agregate = {
    delegate: _delegate,
    foo: function () {
        return this.delegate.foo.call(this);
    }
};

agregate.foo(); // delegate.foo

agregate.delegate = {
    foo: function () {
        alert('foo from new delegate');
    }
};

agregate.foo(); // foo from new delegate

```

这种对象关系称为 “has-a” ，而集成是 “is-a ”的关系。

由于显示组合的缺乏（与继承相比的灵活性），增加中间代码也是可以的。

AOP特性

作为面向方面的一个功能，可以使用function decorators。ECMA-262-3规格没有明确定义的 “function decorators” 的概念（和Python相对，这个词是在Python官方定义了）。不过，拥有函数式参数的函数在某些方面是可以装饰和激活的（通过应用所谓的建议）：

最简单的装饰者例子：

```

function checkDecorator(originalFunction) {
    return function () {
        if (fooBar !== 'test') {
            alert('wrong parameter');
            return false;
        }
        return originalFunction();
    };
}

function test() {
    alert('test function');
}

```

```
var testWithCheck = checkDecorator(test);
var fooBar = false;

test(); // 'test function'
testWithCheck(); // 'wrong parameter'

fooBar = 'test';
test(); // 'test function'
testWithCheck(); // 'test function'
```

结论

在这篇文章，我们理清了OOP的概论（我希望这些资料已经对你有用了），下一章节我们将继续面向对象编程之ECMAScript的实现。

其它参考

- [Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems \(by Henry Lieberman\);](#)
- [Prototype-based programming;](#)
- [Class;](#)
- [Object-oriented programming;](#)
- [Abstraction;](#)
- [Encapsulation;](#)
- [Polymorphism;](#)
- [Inheritance;](#)
- [Multiple inheritance;](#)
- [Mixin;](#)
- [Trait;](#)
- [Interface;](#)
- [Abstract class;](#)
- [Object composition;](#)
- [Aspect-oriented programming;](#)
- [Dynamic programming language.](#)

(18) 面向对象编程之ECMAScript实现

介绍

本章是关于ECMAScript面向对象实现的第2篇，第1篇我们讨论的是概论和ECMAScript的比较，如果你还没有读第1篇，在进行本章之前，我强烈建议你先读一下第1篇，因为本篇实在太长了（35页）。

英文原文：<http://dmitrysoshnikov.com/ecmascript/chapter-7-2-oop-ecmascript-implementation/>

注：由于篇幅太长了，难免出现错误，时刻保持修正中。

在概论里，我们延伸到了ECMAScript，现在，当我们知道它OOP实现时，我们再来准确定义一下：

ECMAScript is an object-oriented programming language supporting delegating inheritance based on prototypes.

ECMAScript是一种面向对象语言，支持基于原型的委托式继承。

我们将从最基本的数据类型来分析，首先要了解的是ECMAScript用原始值（primitive values）和对象（objects）来区分实体，因此有些文章里说的“在JavaScript里，一切都是对象”是错误的（不完全对），原始值就是我们这里要讨论的一些数据类型。

数据类型

虽然ECMAScript是可以动态转化类型的动态弱类型语言，它还是有数据类型的。也就是说，一个对象要属于一个实实在在的类型。

标准规范里定义了9种数据类型，但只有6种是在ECMAScript程序里可以直接访问的，它们是：Undefined、Null、Boolean、String、Number、Object。

另外3种类型只能在实现级别访问（ECMAScript对象是不能使用这些类型的）并用于规范来解释一些操作行为、保存中间值。这3种类型是：Reference、List和Completion。

因此，Reference是用来解释delete、typeof、this这样的操作符，并且包含一个基对象和一个属性名称；List描述的是参数列表的行为（在new表达式和函数调用的时候）；Completion是用来解释行为break、continue、return和throw语句的。

原始值类型

回头来看6中用于ECMAScript程序的数据类型，前5种是原始值类型，包括Undefined、Null、Boolean、String、Number、Object。

原始值类型例子：

```
var a = undefined;
var b = null;
var c = true;
var d = 'test';
var e = 10;
```

这些值是在底层上直接实现的，他们不是object，所以没有原型，没有构造函数。

大叔注：这些原生值和我们平时用的(Boolean、String、Number、Object)虽然名字上相似，但不是同一个东西。所以typeof(true)和typeof(Boolean)结果是不一样的，因为typeof(Boolean)的结果是function，所以函数Boolean、String、Number是有原型的（下面的读写属性章节也会提到）。

想知道数据是哪种类型用typeof是最好不过了，有个例子需要注意一下，如果用typeof来判断null的类型，结果是object，为什么呢？因为null的类型是定义为Null的。

```
alert(typeof null); // "object"
```

显示"object"原因是因为规范就是这么规定的：对于Null值的typeof字符串值返回"object "。

规范没有想象解释这个，但是Brendan Eich (JavaScript发明人)注意到null相对于undefined大多数都是用于对象出现的地方，例如设置一个对象为空引用。但是有些文档里有些气人将之归结为bug，而且将该bug放在Brendan Eich也参与讨论的bug列表里，结果就是任其自然，还是把typeof null的结果设置为object（尽管262-3的标准是定义null的类型是Null，262-5已经将标准修改为null的类型是object了）。

Object类型

接着，Object类型（不要和Object构造函数混淆了，现在只讨论抽象类型）是描述ECMAScript对象的唯一一个数据类型。

Object is an unordered collection of key-value pairs.

对象是一个包含key-value对的无序集合

对象的key值被称为属性，属性是原始值和其他对象的容器。如果属性的值是函数我们称它为方法。

例如：

```
var x = { // 对象"x"有3个属性: a, b, c
  a: 10, // 原始值
  b: {z: 100}, // 对象"b"有一个属性z
  c: function () { // 函数(方法)
    alert('method x.c');
  }
};

alert(x.a); // 10
alert(x.b); // [object Object]
alert(x.b.z); // 100
x.c(); // 'method x.c'
```

动态性

正如我们在第17章中指出的，ES中的对象是完全动态的。这意味着，在程序执行的时候我们可以任意地添加，修改或删除对象的属性。

例如：

```
var foo = {x: 10};

// 添加新属性
foo.y = 20;
console.log(foo); // {x: 10, y: 20}

// 将属性值修改为函数
foo.x = function () {
  console.log('foo.x');
};

foo.x(); // 'foo.x'

// 删除属性
delete foo.x;
console.log(foo); // {y: 20}
```

有些属性不能被修改——（只读属性、已删除属性或不可配置的属性）。我们将稍后在属性特性里讲解。

另外，ES5规范规定，静态对象不能扩展新的属性，并且它的属性页不能删除或者修改。他们是所谓的冻结对象，可以通过应用Object.freeze(o)方法得到。

```
var foo = {x: 10};

// 冻结对象
Object.freeze(foo);
console.log(Object.isFrozen(foo)); // true

// 不能修改
foo.x = 100;

// 不能扩展
foo.y = 200;

// 不能删除
delete foo.x;

console.log(foo); // {x: 10}
```

在ES5规范里，也使用Object.preventExtensions(o)方法防止扩展，或者使用Object.defineProperty(o)方法来定义属性：

```
var foo = {x : 10};

Object.defineProperty(foo, "y", {
  value: 20,
  writable: false, // 只读
  configurable: false // 不可配置
});

// 不能修改
foo.y = 200;

// 不能删除
delete foo.y; // false

// 防治扩展
Object.preventExtensions(foo);
console.log(Object.isExtensible(foo)); // false

// 不能添加新属性
foo.z = 30;

console.log(foo); {x: 10, y: 20}
```

内置对象、原生对象及宿主对象

有必要需要注意的是规范还区分了这内置对象、元素对象和宿主对象。

内置对象和元素对象是被ECMAScript规范定义和实现的，两者之间的差异微不足道。所有ECMAScript实现的对象都是原生对象（其中一些是内置对象、一些在程序执行的时候创建，例如用户自定义对象）。内置对象是原生对象的一个子集、是在程序开始之前内置到ECMAScript里的（例如，parseInt, Match等）。所有的宿主对象是由宿主环境提供的，通常是浏览器，并可能包括如window、alert等。

注意，宿主对象可能是ES自身实现的，完全符合规范的语义。从这点来说，他们能称为“原生宿主”对象（尽快很理论），不过规范没有定义“原生宿主”对象的概念。

Boolean，String和Number对象

另外，规范也定义了一些原生的特殊包装类，这些对象是：

1. 布尔对象
2. 字符串对象
3. 数字对象

这些对象的创建，是通过相应的内置构造器创建，并且包含原生值作为其内部属性，这些对象可以转换省原始值，反之亦然。

```
var c = new Boolean(true);
var d = new String('test');
var e = new Number(10);

// 转换成原始值
// 使用不带new关键字的函数
c = Boolean(c);
d = String(d);
e = Number(e);

// 重新转换成对象
c = Object(c);
d = Object(d);
e = Object(e);
```

此外，也有对象是由特殊的内置构造函数创建：Function（函数对象构造器）、Array（数组构造器）、RegExp（正则表达式构造器）、Math（数学模块）、Date（日期的构造器）等等，这些对象也是Object对象类型的值，他们彼此的区别是由内部属性管理的，我们在下面讨论这些内容。

字面量Literal

对于三个对象的值：对象（object）、数组（array）和正则表达式（regular expression），他们分别有简写的标示符称为：对象初始化器、数组初始化器、和正则表达式初始化器：

```
// 等价于new Array(1, 2, 3);
// 或者array = new Array();
// array[0] = 1;
// array[1] = 2;
// array[2] = 3;
var array = [1, 2, 3];

// 等价于
// var object = new Object();
// object.a = 1;
// object.b = 2;
// object.c = 3;
var object = {a: 1, b: 2, c: 3};

// 等价于new RegExp("^\\d+$", "g")
var re = /^\\d+$/g;
```

注意，如果上述三个对象进行重新赋值名称到新的类型上的话，那随后的实现语义就是按照新赋值的类型来使用，例如在当前的Rhino和老版本SpiderMonkey 1.7的实现上，会成功以new关键字的构造器来创建对象，但有些实现（当前Spider/TraceMonkey）字面量的语义在类型改变以后却不一定改变。

```
var getClass = Object.prototype.toString;

Object = Number;

var foo = new Object;
alert([foo, getClass.call(foo)]); // 0, "[object Number]"

var bar = {};

// Rhino, SpiderMonkey 1.7中 - 0, "[object Number]"
// 其它: still "[object Object]", "[object Object]"
alert([bar, getClass.call(bar)]);

// Array也是一样的效果
Array = Number;

foo = new Array;
alert([foo, getClass.call(foo)]); // 0, "[object Number]"

bar = [];

// Rhino, SpiderMonkey 1.7中 - 0, "[object Number]"
// 其它: still "", "[object Object]"
alert([bar, getClass.call(bar)]);

// 但对RegExp,字面量的语义是不被改变的。 semantics of the literal
// isn't being changed in all tested implementations

RegExp = Number;
```



```
foo = new RegExp;
alert([foo, getClass.call(foo)]); // 0, "[object Number]"

bar = /(?!)/g;
alert([bar, getClass.call(bar)]); // /(?!)/g, "[object RegExp]"
```

正则表达式字面量和RegExp对象

注意，下面2个例子在第三版的规范里，正则表达式的语义都是等价的，regexp字面量只在一句里存在，并且再解析阶段创建，但RegExp构造器创建的却是新对象，所以这可能会导致出一些问题，如lastIndex的值在测试的时候结果是错误的：

```
for (var k = 0; k < 4; k++) {
  var re = /ecma/g;
  alert(re.lastIndex); // 0, 4, 0, 4
  alert(re.test("ecmascript")); // true, false, true, false
}

// 对比

for (var k = 0; k < 4; k++) {
  var re = new RegExp("ecma", "g");
  alert(re.lastIndex); // 0, 0, 0, 0
  alert(re.test("ecmascript")); // true, true, true, true
}
```

注：不过这些问题在第5版的ES规范都已经修正了，不管是基于字面量的还是构造器的，正则都是创建新对象。

关联数组

各种文字静态讨论，JavaScript对象（经常是用对象初始化器{}来创建）被称为哈希表哈希表或其它简单的称谓：哈希（Ruby或Perl里的概念），管理数组（PHP里的概念），词典（Python里的概念）等。

只有这样的术语，主要是因为他们的结构都是相似的，就是使用“键-值”对来存储对象，完全符合“关联数组”或“哈希表”理论定义的数据结构。此外，哈希表抽象数据类型通常是在实现层面使用。

但是，尽管术语上来描述这个概念，但实际上这个是错误，从ECMAScript来看：ECMAScript只有一个对象以及类型以及它的子类型，这和“键-值”对存储没有什么区别，因此在这上面没有特别的概念。因为任何对象的内部属性都可以存储为键-值对：

```

var a = {x: 10};
a['y'] = 20;
a.z = 30;

var b = new Number(1);
b.x = 10;
b.y = 20;
b['z'] = 30;

var c = new Function('');
c.x = 10;
c.y = 20;
c['z'] = 30;

// 等等, 任意对象的子类型"subtype"

```

此外, 由于在ECMAScript中对象可以是空的, 所以"hash"的概念在这里也是不正确的:

```

Object.prototype.x = 10;

var a = {}; // 创建空"hash"

alert(a["x"]); // 10, 但不为空
alert(a.toString()); // function

a["y"] = 20; // 添加新的键值对到 "hash"
alert(a["y"]); // 20

Object.prototype.y = 20; // 添加原型属性

delete a["y"]; // 删除
alert(a["y"]); // 但这里key和value依然有值 - 20

```

请注意, ES5标准可以让我们创建没原型的对象(使用Object.create(null)方法实现)对, 从这个角度来说, 这样的对象可以称之为_哈希表:

```

var aHashTable = Object.create(null);
console.log(aHashTable.toString()); // 未定义

```

此外, 一些属性有特定的getter / setter方法, 所以也可能导致混淆这个概念:

```

var a = new String("foo");
a['length'] = 10;
alert(a['length']); // 3

```

然而, 即使认为“哈希”可能有一个“原型”(例如, 在Ruby或Python里委托哈希对象的
本文档使用 [看云](#) 构建

类)，在ECMAScript里，这个术语也是不对的，因为2个表示法之间没有语义上的区别（即用点表示法a.b和a["b"]表示法）。

在ECMAScript中的“property属性”的概念语义上和“key”、数组索引、方法没有分开的，这里所有对象的属性读写都要遵循统一的规则：检查原型链。

在下面Ruby的例子中，我们可以看到语义上的区别：

```
a = {}
a.class # Hash

a.length # 0

# new "key-value" pair
a['length'] = 10;

# 语义上，用点访问的是属性或方法，而不是key

a.length # 1

# 而索引器访问访问的是hash里的key

a['length'] # 10

# 就类似于在现有对象上动态声明Hash类
# 然后声明新属性或方法

class Hash
  def z
    100
  end
end

# 新属性可以访问

a.z # 100

# 但不是"key"

a['z'] # nil
```

ECMA-262-3标准并没有定义“哈希”（以及类似）的概念。但是，有这样的结构理论的话，那可能以此命名的对象。

对象转换

将对象转化成原始值可以用valueOf方法，正如我们所说的，当函数的构造函数调用做为function（对于某些类型的），但如果不用new关键字就是将对象转化成原始值，就相当于隐式的valueOf方法调用：

```

var a = new Number(1);
var primitiveA = Number(a); // 隐式"valueOf"调用
var alsoPrimitiveA = a.valueOf(); // 显式调用

alert([
  typeof a, // "object"
  typeof primitiveA, // "number"
  typeof alsoPrimitiveA // "number"
]);

```

这种方式允许对象参与各种操作，例如：

```

var a = new Number(1);
var b = new Number(2);

alert(a + b); // 3

// 甚至

var c = {
  x: 10,
  y: 20,
  valueOf: function () {
    return this.x + this.y;
  }
};

var d = {
  x: 30,
  y: 40,
  // 和c的valueOf功能一样
  valueOf: c.valueOf
};

alert(c + d); // 100

```

valueOf的默认值会根据对象的类型改变（如果不被覆盖的话），对某些对象，他返回的是this——例如：Object.prototype.valueOf()，还有计算型的值：Date.prototype.valueOf()返回的是日期时间：

```

var a = {};
alert(a.valueOf() === a); // true, "valueOf"返回this

var d = new Date();
alert(d.valueOf()); // time
alert(d.valueOf() === d.getTime()); // true

```

此外,对象还有一个更原始的代表性——字符串展示。这个toString方法是可靠的,它在某些操作上是自动使用的:

```
var a = {
  valueOf: function () {
    return 100;
  },
  toString: function () {
    return '__test';
  }
};

// 这个操作里, toString方法自动调用
alert(a); // "__test"

// 但是这里, 调用的却是valueOf()方法
alert(a + 10); // 110

// 但, 一旦valueOf删除以后
// toString又可以自动调用了
delete a.valueOf;
alert(a + 10); // "_test10"
```

Object.prototype上定义的toString方法具有特殊意义, 它返回的我们下面将要讨论的内部[[Class]]属性值。

和转化成原始值 (ToPrimitive) 相比, 将值转化成对象类型也有一个转化规范 (ToObject)。

一个显式方法是使用内置的Object构造函数作为function来调用ToObject (有些类似通过new关键字也可以):

```
var n = Object(1); // [object Number]
var s = Object('test'); // [object String]

// 一些类似, 使用new操作符也可以
var b = new Object(true); // [object Boolean]

// 应用参数new Object的话创建的是简单对象
var o = new Object(); // [object Object]

// 如果参数是一个现有的对象
// 那创建的结果就是简单返回该对象
var a = [];
alert(a === new Object(a)); // true
alert(a === Object(a)); // true
```

关于调用内置构造函数，使用还是不适用new操作符没有通用规则，取决于构造函数。例如Array或Function当使用new操作符的构造函数或者不使用new操作符的简单函数使用产生相同的结果的：

```
var a = Array(1, 2, 3); // [object Array]
var b = new Array(1, 2, 3); // [object Array]
var c = [1, 2, 3]; // [object Array]

var d = Function(''); // [object Function]
var e = new Function(''); // [object Function]
```

有些操作符使用的时候，也有一些显示和隐式转化：

```
var a = 1;
var b = 2;

// 隐式
var c = a + b; // 3, number
var d = a + b + '5' // "35", string

// 显式
var e = '10'; // "10", string
var f = +e; // 10, number
var g = parseInt(e, 10); // 10, number

// 等等
```

属性的特性

所有的属性（property）都可以有很多特性（attributes）。

1. {ReadOnly}——忽略向属性赋值的写操作尝试，但只读属性可以由宿主环境行为改变——也就是说不是“恒定值”；
2. {DontEnum}——属性不能被for..in循环枚举
3. {DontDelete}——糊了delete操作符的行为被忽略（即删不掉）；
4. {Internal}——内部属性，没有名字（仅在实现层面使用），ECMAScript里无法访问这样的属性。

注意，在ES5里{ReadOnly}，{DontEnum}和{DontDelete}被重新命名为[[Writable]]，[[Enumerable]]和[[Configurable]]，可以手工通过Object.defineProperty或类似的方法来管理这些属性。

```
var foo = {};
```

```

Object.defineProperty(foo, "x", {
  value: 10,
  writable: true, // 即{ReadOnly} = false
  enumerable: false, // 即{DontEnum} = true
  configurable: true // 即{DontDelete} = false
});

console.log(foo.x); // 10

// 通过descriptor获取特性集attributes
var desc = Object.getOwnPropertyDescriptor(foo, "x");

console.log(desc.enumerable); // false
console.log(desc.writable); // true
// 等等

```

内部属性和方法

对象也可以有内部属性（实现层面的一部分），并且ECMAScript程序无法直接访问（但是下面我们将看到，一些实现允许访问一些这样的属性）。这些属性通过嵌套的中括号[[]]进行访问。我们来看其中的一些，这些属性的描述可以到规范里查阅到。

每个对象都应该实现如下内部属性和方法：

1. `[[Prototype]]`——对象的原型（将在下面详细介绍）
2. `[[Class]]`——字符串对象的一种表示（例如，Object Array，Function Object，Function等）；用来区分对象
3. `[[Get]]`——获得属性值的方法
4. `[[Put]]`——设置属性值的方法
5. `[[CanPut]]`——检查属性是否可写
6. `[[HasProperty]]`——检查对象是否已经拥有该属性
7. `[[Delete]]`——从对象删除该属性
8. `[[DefaultValue]]`返回对象对于的原始值（调用valueOf方法，某些对象可能会抛出TypeError异常）。

通过Object.prototype.toString()方法可以间接得到内部属性[[Class]]的值，该方法应该返回下列字符串：`"[object " + [[Class]] + "]"`。例如：

```

var getClass = Object.prototype.toString;

getClass.call({}); // [object Object]
getClass.call([]); // [object Array]
getClass.call(new Number(1)); // [object Number]
// 等等

```

这个功能通常是用来检查对象用的，但规范上说宿主对象的[[Class]]可以为任意值，包括内置对象的[[Class]]属性的值，所以理论上来看是不能100%来保证准确的。例如，document.childNodes.item(...)方法的[[Class]]属性，在IE里返回"String"，但其它实现里返回的确实"Function"。

```
// in IE - "String", in other - "Function"
alert(getClass.call(document.childNodes.item));
```

构造函数

因此，正如我们上面提到的，在ECMAScript中的对象是通过所谓的构造函数来创建的。

Constructor is a function that creates and initializes the newly created object.

构造函数是一个函数，用来创建并初始化新创建的对象。

对象创建（内存分配）是由构造函数的内部方法[[Construct]]负责的。该内部方法的行为是定义好的，所有的构造函数都是使用该方法来为新对象分配内存的。

而初始化是通过新建对象上下上调用该函数来管理的，这是由构造函数的内部方法[[Call]]来负责的。

注意，用户代码只能在初始化阶段访问，虽然在初始化阶段我们可以返回不同的对象（忽略第一阶段创建的this对象）：

```
function A() {
  // 更新新创建的对象
  this.x = 10;
  // 但返回的是不同的对象
  return [1, 2, 3];
}

var a = new A();
console.log(a.x, a); undefined, [1, 2, 3]
```

引用15章函数——创建函数的算法小节，我们可以看到该函数是一个原生对象，包含[[Construct]]和[[Call]]属性以及显示的prototype原型属性——未来对象的原型（注：NativeObject是对于native object原生对象的约定，在下面的伪代码中使用）。

```
F = new NativeObject();
```



```

F.[[Class]] = "Function"

.... // 其它属性

F.[[Call]] = function> // function自身

F.[[Construct]] = internalConstructor // 普通的内部构造函数

.... // 其它属性

// F构造函数创建的对象原型
__objectPrototype = {};
__objectPrototype.constructor = F // {DontEnum}
F.prototype = __objectPrototype

```

[[Call]] 是除[[Class]]属性（这里等同于"Function"）之外区分对象的主要方式，因此，对象的内部[[Call]]属性作为函数调用。这样的对象用typeof运算操作符的话返回的是"function"。然而它主要是和原生对象有关，有些情况的实现在用typeof获取值的是不一样的，例如：window.alert (...)在IE中的效果：

```

// IE浏览器中 - "Object", "object", 其它浏览器 - "Function", "function"
alert(Object.prototype.toString.call(window.alert));
alert(typeof window.alert); // "Object"

```

内部方法[[Construct]]是通过使用带new运算符的构造函数来激活的，正如我们所说的这个方法负责内存分配和对对象创建的。如果没有参数，调用构造函数的括号也可以省略：

```

function A(x) { // constructor A
  this.x = x || 10;
}

// 不传参数的话，括号也可以省略
var a = new A; // or new A();
alert(a.x); // 10

// 显式传入参数x
var b = new A(20);
alert(b.x); // 20

```

我们也知道，构造函数（初始化阶段）里的this被设置为新创建的对象。

让我们研究一下对象创建的算法。

对象创建的算法

内部方法[[Construct]] 的行为可以描述成如下：

```
F. [[Construct]](initialParameters):
O = new NativeObject();

// 属性[[Class]]被设置为"Object"
O. [[Class]] = "Object"

// 引用F.prototype的时候获取该对象g
var __objectPrototype = F.prototype;

// 如果__objectPrototype是对象, 就:
O. [[Prototype]] = __objectPrototype
// 否则:
O. [[Prototype]] = Object.prototype;
// 这里O. [[Prototype]]是Object对象的原型

// 新创建对象初始化的时候应用了F. [[Call]]
// 将this设置为新创建的对象O
// 参数和F里的initialParameters是一样的
R = F. [[Call]](initialParameters); this === O;
// 这里R是[[Call]]的返回值
// 在JS里看, 像这样:
// R = F.apply(O, initialParameters);

// 如果R是对象
return R
// 否则
return O
```

请注意两个主要特点：

1. 首先，新创建对象的原型是从当前时刻函数的prototype属性获取的（这意味着同一个构造函数创建的两个创建对象的原型可以不同是因为函数的prototype属性也可以不同）。
2. 其次，正如我们上面提到的，如果在对象初始化的时候，[[Call]]返回的是对象，这恰恰是用于整个new操作符的结果：

```
function A() {}
A.prototype.x = 10;

var a = new A();
alert(a.x); // 10 - 从原型上得到

// 设置.prototype属性为新对象
// 为什么显式声明.constructor属性将在下面说明
A.prototype = {
  constructor: A,
  y: 100
};
```

```

var b = new A();
// 对象"b"有了新属性
alert(b.x); // undefined
alert(b.y); // 100 - 从原型上得到

// 但a对象的原型依然可以得到原来的结果
alert(a.x); // 10 - 从原型上得到

function B() {
    this.x = 10;
    return new Array();
}

// 如果"B"构造函数没有返回（或返回this）
// 那么this对象就可以使用，但是下面的情况返回的是array
var b = new B();
alert(b.x); // undefined
alert(Object.prototype.toString.call(b)); // [object Array]

```

让我们来详细了解一下原型

原型

每个对象都有一个原型（一些系统对象除外）。原型通信是通过内部的、隐式的、不可直接访问[[Prototype]]原型属性来进行的，原型可以是一个对象，也可以是null值。

属性构造函数(Property constructor)

上面的例子有2个重要的知识点，第一个是关于函数的constructor属性的prototype属性，在函数创建的算法里，我们知道constructor属性在函数创建阶段被设置为函数的prototype属性，constructor属性的值是函数自身的重要引用：

```

function A() {}
var a = new A();
alert(a.constructor); // function A() {}, by delegation
alert(a.constructor === A); // true

```

通常在这种情况下，存在着一个误区：constructor构造属性作为新创建对象自身的属性是错误的，但是，正如我们所看到的，这个属性属于原型并且通过继承来访问对象。

通过继承constructor属性的实例，可以间接得到的原型对象的引用：

```

function A() {}
A.prototype.x = new Number(10);

var a = new A();
alert(a.constructor.prototype); // [object Object]

```

```

alert(a.x); // 10, 通过原型
// 和a.[[Prototype]].x效果一样
alert(a.constructor.prototype.x); // 10

alert(a.constructor.prototype.x === a.x); // true

```

但请注意，函数的constructor和prototype属性在对象创建以后都可以重新定义的。在这种情况下，对象失去上面所说的机制。如果通过函数的prototype属性去编辑元素的prototype原型的话（添加新对象或修改现有对象），实例上将看到新添加的属性。

然而，如果我们彻底改变函数的prototype属性（通过分配一个新的对象），那原始构造函数的引用就是丢失，这是因为我们创建的对象不包括constructor属性：

```

function A() {}
A.prototype = {
  x: 10
};

var a = new A();
alert(a.x); // 10
alert(a.constructor === A); // false!

```

因此，对函数的原型引用需要手工恢复：

```

function A() {}
A.prototype = {
  constructor: A,
  x: 10
};

var a = new A();
alert(a.x); // 10
alert(a.constructor === A); // true

```

注意虽然手动恢复了constructor属性，和原来丢失的原型相比，{DontEnum}特性没有了，也就是说A.prototype里的for..in循环语句不支持了，不过第5版规范里，通过[[Enumerable]] 特性提供了控制可枚举状态enumerable的能力。

```

var foo = {x: 10};

Object.defineProperty(foo, "y", {
  value: 20,
  enumerable: false // aka {DontEnum} = true
});

```

```

console.log(foo.x, foo.y); // 10, 20

for (var k in foo) {
    console.log(k); // only "x"
}

var xDesc = Object.getOwnPropertyDescriptor(foo, "x");
var yDesc = Object.getOwnPropertyDescriptor(foo, "y");

console.log(
    xDesc.enumerable, // true
    yDesc.enumerable  // false
);

```

显式prototype和隐式[[Prototype]]属性

通常，一个对象的原型通过函数的prototype属性显式引用是不正确的，他引用的是同一个对象，对象的[[Prototype]]属性：

```
a. [[Prototype]] ----> Prototype <---- A.prototype
```

此外，实例的[[Prototype]]值确实是在构造函数的prototype属性上获取的。

然而，提交prototype属性不会影响已经创建对象的原型（只有在构造函数的prototype属性改变的时候才会影响到），就是说新创建的对象才有新的原型，而已创建对象还是引用到原来的旧原型（这个原型已经不能被再被修改了）。

```

// 在修改A.prototype原型之前的情况
a. [[Prototype]] ----> Prototype <---- A.prototype

// 修改之后
A.prototype ----> New prototype // 新对象会拥有这个原型
a. [[Prototype]] ----> Prototype // 引导的原来的原型上

```

例如：

```

function A() {}
A.prototype.x = 10;

var a = new A();
alert(a.x); // 10

A.prototype = {
    constructor: A,
    x: 20
    y: 30
}

```

```

};

// 对象a是通过隐式的[[Prototype]]引用从原型的prototype上获取的值
alert(a.x); // 10
alert(a.y) // undefined

var b = new A();

// 但新对象是从新原型上获取的值
alert(b.x); // 20
alert(b.y) // 30

```

因此，有的文章说“动态修改原型将影响所有的对象都会拥有新的原型”是错误的，新原型仅仅在原型修改以后的新创建对象上生效。

这里的主要规则是：对象的原型是对象的创建的时候创建的，并且在此之后不能修改为新的对象，如果依然引用到同一个对象，可以通过构造函数的显式prototype引用，对象创建以后，只能对原型的属性进行添加或修改。

非标准的proto属性

然而，有些实现（例如SpiderMonkey），提供了不标准的proto显式属性来引用对象的原型：

```

function A() {}
A.prototype.x = 10;

var a = new A();
alert(a.x); // 10

var __newPrototype = {
  constructor: A,
  x: 20,
  y: 30
};

// 引用到新对象
A.prototype = __newPrototype;

var b = new A();
alert(b.x); // 20
alert(b.y); // 30

// "a"对象使用的依然是旧的原型
alert(a.x); // 10
alert(a.y); // undefined

// 显式修改原型
a.__proto__ = __newPrototype;

```

```
// 现在"a"对象引用的是新对象
alert(a.x); // 20
alert(a.y); // 30
```

注意，ES5提供了Object.getPrototypeOf(O)方法，该方法直接返回对象的[[Prototype]]属性——实例的初始原型。然而，和proto相比，它只是getter，它不允许set值。

```
var foo = {};
Object.getPrototypeOf(foo) == Object.prototype; // true
```

对象独立于构造函数

因为实例的原型独立于构造函数和构造函数的prototype属性，构造函数完成了自己的工作（创建对象）以后可以删除。原型对象通过引用[[Prototype]]属性继续存在：

```
function A() {}
A.prototype.x = 10;

var a = new A();
alert(a.x); // 10

// 设置A为null - 显示引用构造函数
A = null;

// 但如果.constructor属性没有改变的话，
// 依然可以通过它创建对象
var b = new a.constructor();
alert(b.x); // 10

// 隐式的引用也删除掉
delete a.constructor.prototype.constructor;
delete b.constructor.prototype.constructor;

// 通过A的构造函数再也不能创建对象了
// 但这2个对象依然有自己的原型
alert(a.x); // 10
alert(b.x); // 10
```

instanceof操作符的特性

我们是通过构造函数的prototype属性来显示引用原型的，这和instanceof操作符有关。该操作符是和原型链一起工作的，而不是构造函数，考虑到这一点，当检测对象的时候往往会有误解：

```
if (foo instanceof Foo) {
    ...
}
```

这不是用来检测对象foo是否是用Foo构造函数创建的，所有instanceof运算符只需要一个对象属性——foo.[[Prototype]]，在原型链中从Foo.prototype开始检查其是否存在。instanceof运算符是通过构造函数里的内部方法[[HasInstance]]来激活的。

让我们来看看这个例子：

```
function A() {}
A.prototype.x = 10;

var a = new A();
alert(a.x); // 10

alert(a instanceof A); // true

// 如果设置原型为null
A.prototype = null;

// ... "a"依然可以通过a.[[Prototype]]访问原型
alert(a.x); // 10

// 不过，instanceof操作符不能再正常使用了
// 因为它是从构造函数的prototype属性来实现的
alert(a instanceof A); // 错误，A.prototype不是对象
```

另一方面，可以由构造函数来创建对象，但如果对象的[[Prototype]]属性和构造函数的prototype属性的值设置的是一样的话，instanceof检查的时候会返回true：

```
function B() {}
var b = new B();

alert(b instanceof B); // true

function C() {}

var __proto = {
  constructor: C
};

C.prototype = __proto;
b.__proto__ = __proto;

alert(b instanceof C); // true
alert(b instanceof B); // false
```

原型可以存放方法并共享属性

大部分程序里使用原型是用来存储对象的方法、默认状态和共享对象的属性。

事实上，对象可以拥有自己的状态，但方法通常是一样的。因此，为了内存优化，方法通常是在原型里定义的。这意味着，这个构造函数创建的所有实例都可以共享找个方法。

```
function A(x) {
    this.x = x || 100;
}

A.prototype = (function () {

    // 初始化上下文
    // 使用额外的对象

    var _someSharedVar = 500;

    function _someHelper() {
        alert('internal helper: ' + _someSharedVar);
    }

    function method1() {
        alert('method1: ' + this.x);
    }

    function method2() {
        alert('method2: ' + this.x);
        _someHelper();
    }

    // 原型自身
    return {
        constructor: A,
        method1: method1,
        method2: method2
    };

})();

var a = new A(10);
var b = new A(20);

a.method1(); // method1: 10
a.method2(); // method2: 10, internal helper: 500

b.method1(); // method1: 20
b.method2(); // method2: 20, internal helper: 500

// 2个对象使用的是原型里相同的方法
alert(a.method1 === b.method1); // true
alert(a.method2 === b.method2); // true
```

读写属性

正如我们提到，读取和写入属性值是通过内部的[[Get]]和[[Put]]方法。这些内部方法是通过

属性访问器激活的：点标记法或者索引标记法：

```
// 写入
foo.bar = 10; // 调用了[[Put]]

console.log(foo.bar); // 10, 调用了[[Get]]
console.log(foo['bar']); // 效果一样
```

让我们用伪代码来看一下这些方法是如何工作的：

[[Get]]方法

[[Get]]也会从原型链中查询属性，所以通过对象也可以访问原型中的属性。

```
O. [[Get]](P):

// 如果是自己的属性，就返回
if (O.hasOwnProperty(P)) {
    return O.P;
}

// 否则，继续分析原型
var __proto = O. [[Prototype]];

// 如果原型是null，返回undefined
// 这是可能的：最顶层Object.prototype. [[Prototype]]是null
if (__proto === null) {
    return undefined;
}

// 否则，对原型链递归调用[[Get]]，在各层的原型中查找属性
// 直到原型为null
return __proto. [[Get]](P)
```

请注意，因为[[Get]]在如下情况也会返回undefined：

```
if (window.someObject) {
    ...
}
```

这里，在window里没有找到someObject属性，然后会在原型里找，原型的原型里找，以此类推，如果都找不到，按照定义就返回undefined。

注意：in操作符也可以负责查找属性（也会查找原型链）：

```
if ('someObject' in window) {
```

```
    ...
}
```

这有助于避免一些特殊问题：比如即便someObject存在，在someObject等于false的时候，第一轮检测就通不过。

[[Put]]方法

[[Put]]方法可以创建、更新对象自身的属性，并且掩盖原型里的同名属性。

```
O.[[Put]](P, V):

// 如果不能给属性写值，就退出
if (!O.[[CanPut]](P)) {
    return;
}

// 如果对象没有自身的属性，就创建它
// 所有的attributes特性都是false
if (!O.hasOwnProperty(P)) {
    createNewProperty(O, P, attributes: {
        ReadOnly: false,
        DontEnum: false,
        DontDelete: false,
        Internal: false
    });
}

// 如果属性存在就设置值，但不改变attributes特性
O.P = V

return;
```

例如：

```
Object.prototype.x = 100;

var foo = {};
console.log(foo.x); // 100, 继承属性

foo.x = 10; // [[Put]]
console.log(foo.x); // 10, 自身属性

delete foo.x;
console.log(foo.x); // 重新是100, 继承属性
```

请注意，不能掩盖原型里的只读属性，赋值结果将忽略，这是由内部方法[[CanPut]]控制的。

```
// 例如，属性length是只读的，我们来掩盖一下length试试

function SuperString() {
  /* nothing */
}

SuperString.prototype = new String("abc");

var foo = new SuperString();

console.log(foo.length); // 3, "abc"的长度

// 尝试掩盖
foo.length = 5;
console.log(foo.length); // 依然是3
```

但在ES5的严格模式下，如果掩盖只读属性的话，会保存TypeError错误。

属性访问器

内部方法[[Get]]和[[Put]]在ECMAScript里是通过点符号或者索引法来激活的，如果属性标示符是合法的名字的话，可以通过“.”来访问，而索引方运行动态定义名称。

```
var a = {testProperty: 10};

alert(a.testProperty); // 10, 点
alert(a['testProperty']); // 10, 索引

var propertyName = 'Property';
alert(a['test' + propertyName]); // 10, 动态属性通过索引的方式
```

这里有一个非常重要的特性——属性访问器总是使用ToObject规范来对待“.”左边的值。这种隐式转化和这句“在JavaScript中一切都是对象”有关系，（然而，当我们已经知道了，JavaScript里不是所有的值都是对象）。

如果对原始值进行属性访问器取值，访问之前会先对原始值进行对象包装（包括原始值），然后通过包装的对象进行访问属性，属性访问以后，包装对象就会被删除。

例如：

```
var a = 10; // 原始值

// 但是可以访问方法（就像对象一样）
alert(a.toString()); // "10"

// 此外，我们可以在a上创建一个心属性
a.test = 100; // 好像是没问题的
```

本文档使用 [看云](#) 构建

```
// 但, [[Get]]方法没有返回该属性的值, 返回的却是undefined  
alert(a.test); // undefined
```

那么, 为什么整个例子中的原始值可以访问toString方法, 而不能访问新创建的test属性呢?

答案很简单:

首先, 正如我们所说, 使用属性访问器以后, 它已经不是原始值了, 而是一个包装过的中间对象 (整个例子是使用new Number(a)), 而toString方法这时候是通过原型链查找到的:

```
// 执行a.toString()的原理:  
  
1\ . wrapper = new Number(a);  
2\ . wrapper.toString(); // "10"  
3\ . delete wrapper;
```

接下来, [[Put]]方法创建新属性时候, 也是通过包装装的对象进行的:

```
// 执行a.test = 100的原理:  
  
1\ . wrapper = new Number(a);  
2\ . wrapper.test = 100;  
3\ . delete wrapper;
```

我们看到, 在第3步的时候, 包装的对象以及删除了, 随着新创建的属性页被删除了——删除包装对象本身。

然后使用[[Get]]获取test值的时候, 再一次创建了包装对象, 但这时候包装的对象已经没有test属性了, 所以返回的是undefined:

```
// 执行a.test的原理:  
  
1\ . wrapper = new Number(a);  
2\ . wrapper.test; // undefined
```

这种方式解释了原始值的读取方式, 另外, 任何原始值如果经常用在访问属性的话, 时间效率考虑, 都是直接用一个对象替代它; 与此相反, 如果不经常访问, 或者只是用于计算的话, 到可以保留这种形式。

继承

我们知道，ECMAScript是使用基于原型的委托式继承。链和原型在原型链里已经提到过了。其实，所有委托的实现和原型链的查找分析都浓缩到[[Get]]方法了。

如果你完全理解[[Get]]方法，那JavaScript中的继承这个问题将不解自答了。

经常在论坛上谈论JavaScript中的继承时，我都是用一行代码来展示，事实上，我们不需要创建任何对象或函数，因为该语言已经是基于继承的了，代码如下：

```
alert(1..toString()); // "1"
```

我们已经知道了[[Get]]方法和属性访问器的原理了，我们来看看都发生了什么：

1. 首先，从原始值1，通过new Number(1)创建包装对象
2. 然后toString方法是从这个包装对象上继承得到的

为什么是继承的？因为在ECMAScript中的对象可以有自己的属性，包装对象在这种情况下没有toString方法。因此它是从原理里继承的，即Number.prototype。

注意有个微妙的地方，在上面的例子中的两个点不是一个错误。第一点是代表小数部分，第二个才是一个属性访问器：

```
1.toString(); // 语法错误！
(1).toString(); // OK
1..toString(); // OK
1['toString'](); // OK
```

原型链

让我们展示如何为用户定义对象创建原型链，非常简单：

```
function A() {
  alert('A.[[Call]] activated');
  this.x = 10;
}
A.prototype.y = 20;

var a = new A();
alert([a.x, a.y]); // 10 (自身), 20 (继承)

function B() {}

// 最近的原型链方式就是设置对象的原型为另外一个新对象
B.prototype = new A();
```

```
// 修复原型的constructor属性, 否则的话是A了
B.prototype.constructor = B;

var b = new B();
alert([b.x, b.y]); // 10, 20, 2个都是继承的

// [[Get]] b.x:
// b.x (no) -->
// b.[[Prototype]].x (yes) - 10

// [[Get]] b.y
// b.y (no) -->
// b.[[Prototype]].y (no) -->
// b.[[Prototype]].[[Prototype]].y (yes) - 20

// where b.[[Prototype]] === B.prototype,
// and b.[[Prototype]].[[Prototype]] === A.prototype
```

这种方法有两个特性：

首先，B.prototype将包含x属性。乍一看这可能不对，你可能会想x属性是在A里定义的并且B构造函数也是这样期望的。尽管原型继承正常情况是没问题的，但B构造函数有时候可能不需要x属性，与基于class的继承相比，所有的属性都复制到后代子类里了。

尽管如此，如果有需要（模拟基于类的继承）将x属性赋给B构造函数创建的对象上，有一些方法，我们后来展示其中一种方式。

其次，这不是一个特征而是缺点——子类原型创建的时候，构造函数的代码也执行了，我们可以看到消息"A.[[Call]] activated"显示了两次——当用A构造函数创建对象赋给B.prototype属性的时候，另外一场是a对象创建自身的时候！

下面的例子比较关键，在父类的构造函数抛出的异常：可能实际对象创建的时候需要检查吧，但很明显，同样的case，也就是就是使用这些父对象作为原型的时候就会出错。

```
function A(param) {
  if (!param) {
    throw 'Param required';
  }
  this.param = param;
}
A.prototype.x = 10;

var a = new A(20);
alert([a.x, a.param]); // 10, 20

function B() {}
B.prototype = new A(); // Error
```

此外，在父类的构造函数有太多代码的话也是一种缺点。

解决这些“功能”和问题，程序员使用原型链的标准模式（下面展示），主要目的就是在中间包装构造函数的创建，这些包装构造函数的链里包含需要的原型。

```
function A() {
    alert('A.[[Call]] activated');
    this.x = 10;
}
A.prototype.y = 20;

var a = new A();
alert([a.x, a.y]); // 10 (自身), 20 (集成)

function B() {
    // 或者使用A.apply(this, arguments)
    B.superproto.constructor.apply(this, arguments);
}

// 继承：通过空的中间构造函数将原型连在一起
var F = function () {};
F.prototype = A.prototype; // 引用
B.prototype = new F();
B.superproto = A.prototype; // 显示引用到另外一个原型上, "sugar"

// 修复原型的constructor属性，否则的就是A了
B.prototype.constructor = B;

var b = new B();
alert([b.x, b.y]); // 10 (自身), 20 (集成)
```

注意，我们在b实例上创建了自己的x属性，通过B.superproto.constructor调用父构造函数来引用新创建对象的上下文。

我们也修复了父构造函数在创建子原型的时候不需要的调用，此时，消息"A.[[Call]] activated"在需要的时候才会显示。

为了在原型链里重复相同的行为（中间构造函数创建，设置superproto，恢复原始构造函数），下面的模板可以封装成一个非常方面的工具函数，其目的是连接原型的时候不是根据构造函数的实际名称。

```
function inherit(child, parent) {
    var F = function () {};
    F.prototype = parent.prototype;
    child.prototype = new F();
    child.prototype.constructor = child;
    child.superproto = parent.prototype;
```



```
    return child;
}
```

因此，继承：

```
function A() {}
A.prototype.x = 10;

function B() {}
inherit(B, A); // 连接原型

var b = new B();
alert(b.x); // 10, 在A.prototype查找到
```

也有很多语法形式（包装而成），但所有的语法行都是为了减少上述代码里的行为。

例如，如果我们把中间的构造函数放到外面，就可以优化前面的代码（因此，只有一个函数被创建），然后重用它：

```
var inherit = (function(){
    function F() {}
    return function (child, parent) {
        F.prototype = parent.prototype;
        child.prototype = new F;
        child.prototype.constructor = child;
        child.superproto = parent.prototype;
        return child;
    };
})();
```

由于对象的真实原型是[[Prototype]]属性，这意味着F.prototype可以很容易修改和重用，因为通过new F创建的child.prototype可以从child.prototype的当前值里获取[[Prototype]]：

```
function A() {}
A.prototype.x = 10;

function B() {}
inherit(B, A);

B.prototype.y = 20;

B.prototype.foo = function () {
    alert("B#foo");
};

var b = new B();
alert(b.x); // 10, 在A.prototype里查到
```

```
function C() {}
inherit(C, B);

// 使用"superproto"语法糖
// 调用父原型的同名方法

C.prototype.foo = function () {
  C.superproto.foo.call(this);
  alert("C#foo");
};

var c = new C();
alert([c.x, c.y]); // 10, 20

c.foo(); // B#foo, C#foo
```

注意，ES5为原型链标准化了这个工具函数，那就是Object.create方法。ES3可以使用以下方式实现：

```
Object.create ||
Object.create = function (parent, properties) {
  function F() {}
  F.prototype = parent;
  var child = new F;
  for (var k in properties) {
    child[k] = properties[k].value;
  }
  return child;
}

// 用法
var foo = {x: 10};
var bar = Object.create(foo, {y: {value: 20}});
console.log(bar.x, bar.y); // 10, 20
```

此外，所有模仿现在基于类的经典继承方式都是根据这个原则实现的，现在可以看到，它实际上不是基于类的继承，而是连接原型的一个很方便的代码重用。

结论

本章内容已经很充分和详细了，希望这些资料对你有用，并且消除你对ECMAScript的疑问，如果你有任何问题，请留言，我们一起讨论。

其它参考

- 4.2 — [Language Overview](#);
- 4.3 — [Definitions](#);

- 7.8.5 — [Regular Expression Literals](#);
- 8 — [Types](#);
- 9 — [Type Conversion](#);
- 11.1.4 — [Array Initialiser](#);
- 11.1.5 — [Object Initialiser](#);
- 11.2.2 — [The new Operator](#);
- 13.2.1 — [\[\[Call\]\]](#);
- 13.2.2 — [\[\[Construct\]\]](#);
- 15 — [Native ECMAScript Objects](#).

(19) 求值策略

介绍

本章，我们将讲解在ECMAScript向函数function传递参数的策略。

计算机科学里对这种策略一般称为“evaluation strategy”（大叔注：有的人说翻译成求值策略，有的人翻译成赋值策略，通看下面的内容，我觉得称为赋值策略更为恰当，anyway，标题还是写成大家容易理解的求值策略吧），例如在编程语言为求值或者计算表达式设置规则。向函数传递参数的策略是一个特殊的case。

<http://dmitrysoshnikov.com/ecmascript/chapter-8-evaluation-strategy/>

写这篇文章的原因是因为论坛上有人要求准确解释一些传参的策略，我们这里给出了相应的定义，希望对大家有所帮助。

很多程序员都确信在JavaScript中（甚至其它一些语言），对象是按引用传参，而原始值类型按值传参，此外，很多文章都说到这个“事实”，但有多人真正理解这个术语，而且又有多少是正确的？我们本篇讲逐一讲解。

一般理论

需要注意到，在赋值理论里一般有2中赋值策略：严格——意思是说参数在进入程序之前是经过计算过的；非严格——意思是参数的计算是根据计算要求才去计算（也就是相当于延迟计算）。

然后，这里我们考虑基本的函数传参策略，从ECMAScript出发点来说是非常重要的。首先需要注意的是，在ECMAScript中（甚至其他的语言如，C，JAVA，Python和Ruby中）都使用了严格的参数传递策略。

另外传递参数的计算顺序也是很重要的——在ECMAScript是左到右，而且其它语言实现的反省顺序（从右向做）也是可以用的。

严格的传参策略也分为几种子策略，其中最重要的一些策略我们在本章详细讨论。

下面讨论的策略不是全部都用在ECMAScript中，所以在讨论这些策略的具体行为的时候，我们使用了伪代码来展示。

按值传递

按值传递，很多开发人员都很了解了，参数的值是调用者传递的对象值的拷贝(copy of

本文档使用 看云 构建

value)，函数内部改变参数的值不会影响到外面的对象（该参数在外面的值），一般来说，是重新分配了新内存(我们不关注分配内存是怎么实现的——也是是栈也许是动态内存分配)，该新内存块的值是外部对象的拷贝，并且它的值是用到函数内部的。

```
bar = 10

procedure foo(barArg):
  barArg = 20;
end

foo(bar)

// foo内部改变值不会影响内部的bar的值
print(bar) // 10
```

但是，如果该函数的参数不是原始值而是复杂的结构对象是时候，将带来很大的性能问题，C++就有这个问题，将结构作为值传进函数的时候——就是完整的拷贝。

我们来给一个一般的例子，用下面的赋值策略来检验一下，想想一下一个函数接受2个参数，第1个参数是对象的值，第2个是个布尔型的标记，用来标记是否完全修改传入的对象（给对象重新赋值），还是只修改该对象的一些属性。

```
// 注：以下都是伪代码，不是JS实现
bar = {
  x: 10,
  y: 20
}

procedure foo(barArg, isFullChange):

  if isFullChange:
    barArg = {z: 1, q: 2}
    exit
  end

  barArg.x = 100
  barArg.y = 200

end

foo(bar)

// 按值传递，外部的对象不被改变
print(bar) // {x: 10, y: 20}

// 完全改变对象（赋新值）
foo(bar, true)
```

```
//也没有改变
print(bar) // {x: 10, y: 20}, 而不是{z: 1, q: 2}
```

按引用传递

另外一个众所周知的按引用传递接收的不是值拷贝，而是对象的隐式引用，如该对象在外部的直接引用地址。函数内部对参数的任何改变都是影响该对象在函数外部的值，因为两者引用的是同一个对象，也就是说：这时候参数就相当于外部对象的一个别名。

伪代码：

```
procedure foo(barArg, isFullChange):
    if isFullChange:
        barArg = {z: 1, q: 2}
        exit
    end

    barArg.x = 100
    barArg.y = 200

end

// 使用和上例相同的对象
bar = {
    x: 10,
    y: 20
}

// 按引用调用的结果如下：
foo(bar)

// 对象的属性值已经被改变了
print(bar) // {x: 100, y: 200}

// 重新赋新值也影响到了该对象
foo(bar, true)

// 此刻该对象已经是一个新对象了
print(bar) // {z: 1, q: 2}
```

该策略可以更有效地传递复杂对象，例如带有大批量属性的大结构对象。

按共享传递 (Call by sharing)

上面2个策略大家都是知道的，但这里要讲的一个策略可能大家不太了解（其实是学术上的策略）。但是，我们很快就会看到这正是它在ECMAScript中的参数传递战略中起着关键作用的策略。

这个策略还有一些代名词：“按对象传递”或“按对象共享传递”。

该策略是1974年由Barbara Liskov为CLU编程语言提出的。

该策略的要点是：函数接收的是对象对于的拷贝（副本），该引用拷贝和形参以及其值相关联。

这里出现的引用，我们不能称之为“按引用传递”，因为函数接收的参数不是直接的对象别名，而是该引用地址的拷贝。

最重要的区别就是：函数内部给参数重新赋新值不会影响到外部的对象（和上例按引用传递的case），但是因为该参数是一个地址拷贝，所以在外面访问和里面访问的都是同一个对象（例如外部的该对象不是想按值传递一样完全的拷贝），改变该参数对象的属性值将会影响到外部的对象。

```
procedure foo(barArg, isFullChange):
    if isFullChange:
        barArg = {z: 1, q: 2}
        exit
    end

    barArg.x = 100
    barArg.y = 200

end

//还是使用这个对象结构
bar = {
    x: 10,
    y: 20
}

// 按贡献传递会影响对象
foo(bar)

// 对象的属性被修改了
print(bar) // {x: 100, y: 200}

// 重新赋值没有起作用
foo(bar, true)

// 依然是上面的值
print(bar) // {x: 100, y: 200}
```

这个处理的假设前提是大多数语言里用到的对象，而不是原始值。

按共享传递是按值传递的特例

按共享传递这个策略很多语言里都使用了：Java, ECMAScript, Python, Ruby, Visual Basic等。此外，Python社区已经使用了这个术语，至于其他语言也可以用这个术语，因为其他的名称往往会让大家感觉到混乱。大多数情况下，例如在Java，ECMAScript或Visual Basic中，这一策略也称之为按值传递——意味着：特殊值——引用拷贝（副本）。

一方面，它是这样的——传递给函数内部用的参数仅仅是绑定值（引用地址）的一个名称，并不会影响外部的对象。

另一方面，如果不深入研究，这些术语真的被认为吃错误的，因为很多论坛都在说如何将对象传递给JavaScript函数）。

一般理论确实有按值传递的说法：但这时候这个值就是我们所说的地址拷贝（副本），因此并没哟破坏规则。

在Ruby中，这个策略称为按引用传递。再说一下：它不是按照大结构的拷贝来传递（例如，不是按值传递），而另一方面，我们没有处理原始对象的引用，并且不能修改它；因此，这个跨术语的概念可能更会造成混乱。

理论里没有像按值传递的特殊case一样来面试按引用传递的特殊case。

但依然有必要了解这些策略在上述提到的技术中（Java, ECMAScript, Python, Ruby, other），实际上——他们用的策略就是按共享传递。

按共享与指针

对于C/C++，这个策略在思想上和按指针值传递是一样的，但有一个重要的区别——该策略可以取消引用指针以及完全改变对象。但在一般情况下，分配一个值（地址）指针到新的内存块（即之前引用的内存块保持不变）；通过指针改变对象属性的话会影响阿东外部对象。

因此，和指针类别，我们可以明显看到，这是按地址值传递。在这种情况下，按共享传递只是“语法糖”，像指针赋值行为一样（但不能取消引用），或者像引用一样修改属性（不需要取消引用操作），有时候，它可以被命名为“安全指针”。

然而，C/C++如果在没有明显指针的解引用的情况下，引用对象属性时，还具有特殊的语法糖：

```
obj->x instead of (*obj).x
```


和C++关系最为紧密的这种意识形态可以从“智能指针”的实现中看到，例如，在 `boost::shared_ptr` 里，重载了赋值操作符以及拷贝构造函数，而且还使用了对象的引用计数器，通过GC删除对象。这种数据类型，甚至有类似的名字- `共享_ptr`。

ECMAScript实现

现在我们知道了ECMAScript中将对象作为参数传递的策略了——按共享传递：修改参数的属性将会影响到外部，而重新赋值将不会影响到外部对象。但是，正如我们上面提到的，其中的ECMAScript开发人员一般都称之为是：按值传递，只不过该值是引用地址的拷贝。

JavaScript发明人布伦丹·艾希也写到了：传递的是引用的拷贝（地址副本）。所以论坛里大家曾说的按值传递，在这种解释下，也是对的。

更确切地说，这种行为可以理解为简单的赋值，我们可以看到，内部是完全不同的对象，只不过引用的是相同的值——也就是地址副本。

ECMAScript代码：

```
var foo = {x: 10, y: 20};
var bar = foo;

alert(bar === foo); // true

bar.x = 100;
bar.y = 200;

alert([foo.x, foo.y]); // [100, 200]
```

即两个标识符（名称绑定）绑定到内存中的同一个对象，`_共享_`这个对象：

```
foo value: addr(0xFF) => {x: 100, y: 200} (address 0xFF) 0xFF)
```

而重新赋值分配，绑定是新的对象标识符（新地址），而不影响已经先前绑定的对象：

```
bar = {z: 1, q: 2};

alert([foo.x, foo.y]); // [100, 200] - 没改变
alert([bar.z, bar.q]); // [1, 2] - 但现在引用的是新对象
```

即现在`foo`和 `bar`，有不同的值和不同的地址：

```
foo value: addr(0xFF) => {x: 100, y: 200} (address 0xFF)
```

```
bar value: addr(0xFA) => {z: 1, q: 2} (address 0xFA)
```

再强调一下，这里所说对象的值是地址（address），而不是对象结构本身，将变量赋值给另外一个变量——是赋值值的引用。因此两个变量引用的是同一个内存地址。下一个赋值却是新地址，是解析与旧对象的地址绑定，然后绑定到新对象的地址上，这就是和按引用传递的最重要区别。

此外，如果只考虑ECMA-262标准所提供的抽象层次，我们在算法里看到的只有“值”这个概念，实现传递的“值”（可以是原始值，也可以是对象），但是按照我们上面的定义，也可以完全称之为“按值传递”，因为引用地址也是值。

然而，为了避免误解（为什么外部对象的属性可以在函数内部改变），这里依然需要考虑实现层面的细节——我们看到的按共享传递，或者换句话讲——按安全指针传递，而安全指针不可能去解除引用和改变对象的，但可以去修改该对象的属性值。

术语版本

让我们来定义ECMAScript中该策略的术语版本。

可以称之为“按值传递”——这里所说的值是一个特殊的case，也就是该值是地址副本（address copy）。从这个层面我们可以说：ECMAScript中除了异常之外的对象都是按值传递的，这实际上是ECMAScript抽象的层面。

或针对这种情况下，专门称之为“按共享传递”，通过这个正好可以看到传统的按值传递和按引用传递的区别，这种情况，可以分成2个种情况：1：原始值按值传递；2：对象按共享传递。

“通过引用类型将对象到函数”这句话和ECMAScript无关，而且它是错误的。

结论

我希望这篇文章有助于宏观上了解更多细节，以及在ECMAScript中的实现。一如既往，如果有任何问题，欢迎讨论。

其它参考

- [Evaluation strategy](#)
- [Call by value](#)
- [Call by reference](#)
- [Call by sharing](#)

(20) 《你真懂JavaScript吗？》答案详解

介绍

昨天发的[《大叔手记 \(19 \) : 你真懂JavaScript吗？》](#)里面的5个题目，有很多回答，发现强人还是很多的，很多人都全部答对了。

今天我们来对这5个题目详细分析一下，希望对大家有所帮助。

注：

问题来自大名鼎鼎的前端架构师Baranovskiy的帖子[《So, you think you know JavaScript?》](#)。

答案也是来自大名鼎鼎的JS牛人Nicholas C. Zakas的帖子[《Answering Baranovskiy's JavaScript quiz》](#)——《JavaScript高级程序设计》一书的原作者
(但题目2的解释貌似有点问题)

OK，我们先看第一题

题目1

```
if (!("a" in window)) {  
    var a = 1;  
}  
alert(a);
```

代码看起来是想说：如果window不包含属性a，就声明一个变量a，然后赋值为1。

你可能认为alert出来的结果是1，然后实际结果是“undefined”。要了解为什么，我们需要知道JavaScript里的3个概念。

首先，所有的全局变量都是window的属性，语句 `var a = 1;`等价于`window.a = 1;` 你可以用如下方式来检测全局变量是否声明：

```
"变量名称" in window
```

第二，所有的变量声明都在范围作用域的顶部，看一下相似的例子：

```
alert("a" in window);  
var a;
```

此时，尽管声明是在alert之后，alert弹出的依然是true，这是因为JavaScript引擎首先会扫描所有的变量声明，然后将这些变量声明移动到顶部，最终的代码效果是这样的：

```
var a;  
alert("a" in window);
```

这样看起来就很容易解释为什么alert结果是true了。

第三，你需要理解该题目的意思是，变量声明被提前了，但变量赋值没有，因为这行代码包括了变量声明和变量赋值。

你可以将语句拆分为如下代码：

```
var a;    //声明  
a = 1;    //初始化赋值
```

当变量声明和赋值在一起用的时候，JavaScript引擎会自动将它分为两部以便将变量声明提前，不将赋值的步骤提前是因为他有可能影响代码执行出不可预期的结果。

所以，知道了这些概念以后，重新回头看一下题目的代码，其实就等价于：

```
var a;  
if (!("a" in window)) {  
    a = 1;  
}  
alert(a);
```

这样，题目的意思就非常清楚了：首先声明a，然后判断a是否存在，如果不存在就赋值为1，很明显a永远在window里存在，这个赋值语句永远不会执行，所以结果是undefined。

大叔注：_提前_这个词语显得有点迷惑了，其实就是执行上下文的关系，因为执行上下文分2个阶段：进入执行上下文和执行代码，在进入执行上下文的时候，创建变量对象VO里已经有了：函数的所有形参、所有的函数声明、所有的变量声明

```
VO(global) = {
```

```
    a: undefined
  }
```

这个时候a已经有了；

然后执行代码的时候才开始走if语句，详细信息请查看[《深入理解JavaScript系列（12）：变量对象（Variable Object）》](#)中的处理上下文代码的2个阶段小节。

大叔注：相信很多人都是认为a在里面不可访问，结果才是undefined的吧，其实是已经有了，只不过初始值是undefined，而不是不可访问。

题目2

```
var a = 1,
    b = function a(x) {
      x && a(--x);
    };
alert(a);
```

这个题目看起来比实际复杂，alert的结果是1；这里依然有3个重要的概念需要我们知道。

首先，在题目1里我们知道了变量声明在进入执行上下文就完成了；第二个概念就是函数声明也是提前的，所有的函数声明都在执行代码之前都已经完成了声明，和变

量声明一样。澄清一下，函数声明是如下这样的代码：

```
function functionName(arg1, arg2){
  //函数体
}
```

如下不是函数，而是函数表达式，相当于变量赋值：

```
var functionName = function(arg1, arg2){
  //函数体
};
```

澄清一下，函数表达式没有提前，就相当于平时的变量赋值。

第三需要知道的是，函数声明会覆盖变量声明，但不会覆盖变量赋值，为了解释这个，我们来看一个例子：

```
function value(){
```

```

    return 1;
}
var value;
alert(typeof value);    //"function"

```

尽快变量声明在下面定义，但是变量value依然是function，也就是说这种情况下，函数声明的优先级高于变量声明的优先级，但如果该变量value赋值了，那结果就完全不一样了：

```

function value(){
    return 1;
}
var value = 1;
alert(typeof value);    //"number"

```

该value赋值以后，变量赋值初始化就覆盖了函数声明。

重新回到题目，这个函数其实是一个有名函数表达式，函数表达式不像函数声明一样可以覆盖变量声明，但你可以注意到，变量b是包含了该函数表达式，而该函数表达式的名字是a；不同的浏览器对a这个名词处理有点不一样，在IE里，会将a认为函数声明，所以它被变量初始化覆盖了，就是说如果调用a(--x)的话就会出错，而其它浏览器在允许在函数内部调用a(--x)，因为这时候a在函数外面依然是数字。基本上，IE里调用b(2)的时候会出错，但其它浏览器则返回undefined。

理解上述内容之后，该题目换成一个更准确和更容易理解的代码应该像这样：

```

var a = 1,
    b = function(x) {
        x && b(--x);
    };
alert(a);

```

这样的话，就很清晰地知道为什么alert的总是1了，详细内容请参考[《深入理解JavaScript系列（2）：揭秘命名函数表达式》](#)中的内容。

大叔注：安装ECMAScript规范，作者对函数声明覆盖变量声明的解释其实不准确的，正确的理解应该是如下：

进入执行上下文：这里出现了名字一样的情况，一个是函数申明，一个是变量申明。那么，根据[深入理解JavaScript系列（12）：变量对象（Variable Object）](#)介绍的，填充VO的顺序是：函数的形参 -> 函数申明 -> 变量申明。

上述例子中，变量a在函数a后面，那么，变量a遇到函数a怎么办呢？还是根据变量对象中介
本文档使用 [看云](#) 构建

绍的，当变量申明遇到VO中已经有同名的时候，不会影响已经存在的属性。而函数表达式不会影响VO的内容，所以b只有在执行的时候才会触发里面的内容。

题目3

```
function a(x) {  
    return x * 2;  
}  
var a;  
alert(a);
```

这个题目就是题目2里的大叔加的注释了，也就是函数声明和变量声明的关系和影响，遇到同名的函数声明，VO不会重新定义，所以这时候全局的VO应该是如下这样的：

```
VO(global) = {  
    a: 引用了函数声明“a”  
}
```

而执行a的时候，相应地就弹出了函数a的内容了。

题目4

```
function b(x, y, a) {  
    arguments[2] = 10;  
    alert(a);  
}  
b(1, 2, 3);
```

关于这个题目，NC搬出了262-3的规范出来解释，其实从《[深入理解JavaScript系列（12）：变量对象（Variable Object）](#)》中的函数上下文中的变量对象一节就可以清楚地知道，活动对象是在进入函数上下文时刻被创建的，它通过函数的arguments属性初始化。arguments属性的值是Arguments对象：

```
AO = {  
    arguments: <Arg0>  
};
```

Arguments对象是活动对象的一个属性，它包括如下属性：

1. callee — 指向当前函数的引用
2. length — 真正传递的参数个数

3. properties-indexes (字符串类型的整数) 属性的值就是函数的参数值(按参数列表从左到右排列)。properties-indexes内部元素的个数等于arguments.length. properties-indexes 的值和实际传递进来的参数之间是共享的。

这个共享其实不是真正的共享一个内存地址，而是2个不同的内存地址，使用JavaScript引擎来保证2个值是随时一样的，当然这也有一个前提，那就是这个索引值要小于你传入的参数个数，也就是说如果你只传入2个参数，而还继续使用arguments[2]赋值的话，就会不一致，例如：

```
function b(x, y, a) {
    arguments[2] = 10;
    alert(a);
}
b(1, 2);
```

这时候因为没传递第三个参数a，所以赋值10以后，alert(a)的结果依然是undefined，而不是10，但如下代码弹出的结果依然是10，因为和a没有关系。

```
function b(x, y, a) {
    arguments[2] = 10;
    alert(arguments[2]);
}
b(1, 2);
```

题目5

```
function a() {
    alert(this);
}
a.call(null);
```

这个题目可以说是最简单的，也是最诡异的，因为如果没学到它的定义的话，打死也不会知道结果的，关于这个题目，我们先来了解2个概念。

首先，就是this值是如何定义的，当一个方法在对象上调用的时候，this就指向到了该对象上，例如：

```
var object = {
    method: function() {
        alert(this === object);    //true
    }
}
```

```
object.method();
```

上面的代码，调用method()的时候this被指向到调用它的object对象上，但在全局作用域里，this是等价于window（浏览器中，非浏览器里等价于global），在如果一个function的定义不是属于一个对象属性的时候（也就是单独定义的函数），函数内部的this也是等价于window的，例如：

```
function method() {
    alert(this === window);    //true
}
method();
```

了解了上述概念之后，我们再来了解一下call()是做什么的，call方法作为一个function执行代表该方法可以让另外一个对象作为调用者来调用，call方法的第一个参数是对象调用者，随后的其它参数是要传给调用method的参数（如果声明了的话），例如：

```
function method() {
    alert(this === window);
}
method();    //true
method.call(document);    //false
```

第一个依然是true没什么好说的，第二个传入的调用对象是document，自然不会等于window，所以弹出了false。

另外，根据ECMAScript262规范规定：如果第一个参数传入的对象调用者是null或者undefined的话，call方法将把全局对象（也就是window）作为this的值。所以，不管你什么时候传入null，其this都是全局对象window，所以该题目可以理解成如下代码：

```
function a() {
    alert(this);
}
a.call(window);
```

所以弹出的结果是[object Window]就很容易理解了。

总结

这5个题目虽然貌似有点偏，但实际上考察的依然是基本概念，只有熟知了这些基本概念才能写出高质量代码。

关于JavaScript的基本核心内容和理解基本上在该系列就到此为止了，接下来的章节除了把五大原则剩余的2篇补全依然，会再加两篇关于DOM的文章，然后就开始转向整理关于JavaScript模式与设计模式相关的文章了（大概10篇左右），随后再会花几个章节来一个实战系列。

更多题目

如果大家有兴趣，可以继续研究下面的一些题目，详细通过这些题目也可以再次加深对JavaScript基础核心特性的理解。

大叔注：这些题目也是来自出这5个题目的人，当然如果你能答对4个及以上并且想拿高工资的话，请联系我。

1. 找出数字数组中最大的元素（使用Math.max函数）
2. 转化一个数字数组为function数组（每个function都弹出相应的数字）
3. 给object数组进行排序（排序条件是元素对象的属性个数）
4. 利用JavaScript打印出Fibonacci数（不使用全局变量）
5. 实现如下语法的功能：`var a = (5).plus(3).minus(6); //2`
6. 实现如下语法的功能：`var a = add(2)(3)(4); //9`

(21) S.O.L.I.D五大原则之接口隔离原则ISP

前言

本章我们要讲解的是S.O.L.I.D五大原则JavaScript语言实现的第4篇，接口隔离原则ISP（The Interface Segregation Principle）。

英文原文：<http://freshbrewedcode.com/derekgreer/2012/01/08/solid-javascript-the-interface-segregation-principle/>

注：这篇文章作者写得比较绕口，所以大叔理解得也比较郁闷，凑合着看吧，别深陷进去了

接口隔离原则的描述是：

Clients should not be forced to depend on methods they do not use.

不应该强迫客户依赖于它们不用的方法。

当用户依赖的接口方法即便只被别的用户使用而自己不用，那它也得实现这些接口，换言之，一个用户依赖了未使用但被其他用户使用的接口，当其他用户修改该接口时，依赖该接口的所有用户都将受到影响。这显然违反了开闭原则，也不是我们所期望的。

接口隔离原则ISP和单一职责有点类似，都是用于聚集功能职责的，实际上ISP可以被理解才具有单一职责的程序转化到一个具有公共接口的对象。

JavaScript接口

JavaScript下我们该如何遵守这个原则呢？毕竟JavaScript没有接口的特性，如果接口就是我们所想的通过某种语言提供的抽象类型来建立contract和解耦的话，那可以说还行，不过JavaScript有另外一种形式的接口。在[Design Patterns – Elements of Reusable Object-Oriented Software](#)一书中我们找到了接口的定义：

一个对象声明的任意一个操作都包含一个操作名称，参数对象和操作的返回值。我们称之为操作符的签名（signature）。

一个对象里声明的所有操作被称为这个对象的接口（interface）。一个对象的接口描绘了所有发生在这个对象上的请求信息。

不管一种语言是否提供一个单独的构造来表示接口，所有的对象都有一个由该对象所有属性和方法组成的隐式接口。参考如下代码：

```

var exampleBinder = {};
exampleBinder.modelObserver = (function() {
    /* 私有变量 */
    return {
        observe: function(model) {
            /* 代码 */
            return newModel;
        },
        onChange: function(callback) {
            /* 代码 */
        }
    }
})();

exampleBinder.viewAdaptor = (function() {
    /* 私有变量 */
    return {
        bind: function(model) {
            /* 代码 */
        }
    }
})();

exampleBinder.bind = function(model) {
    /* 私有变量 */
    exampleBinder.modelObserver.onChange(/* 回调callback */);
    var om = exampleBinder.modelObserver.observe(model);
    exampleBinder.viewAdaptor.bind(om);
    return om;
};

```

上面的exampleBinder类库实现的功能是双向绑定。该类库暴露的公共接口是bind方法，其中bind里用到的关于change通知和view交互的功能分别是由单独的对象modelObserver和viewAdaptor来实现的，这些对象从某种意义上来说就是公共接口bind方法的具体实现。

尽管JavaScript没有提供接口类型来支持对象的contract，但该对象的隐式接口依然能当做一个contract提供给程序用户。

ISP与JavaScript

我们下面讨论的一些小节是JavaScript里关于违反接口隔离原则的影响。正如上面看到的，JavaScript程序里实现接口隔离原则虽然可惜，但是不像静态类型语言那样强大，JavaScript的语言特性有时候会使得所谓的接口搞得有点不粘性。

堕落的实现

在静态类型语言语言里，导致违反ISP原则的一个原因是堕落的实现。在Java和C#里所有的接口里定义的方法都必须实现，如果你只需要其中几个方法，那其他的方法也必须实现（可

以通过空实现或者抛异常的方式)。在JavaScript里，如果只需要一个对象里的某一些接口的话，他也解决不了堕落实现这个问题，虽然不用强制实现上面的接口。但是这种实现依然违反了里氏替换原则。

```
var rectangle = {
  area: function() {
    /* 代码 */
  },
  draw: function() {
    /* 代码 */
  }
};

var geometryApplication = {
  getLargestRectangle: function(rectangles) {
    /* 代码 */
  }
};

var drawingApplication = {
  drawRectangles: function(rectangles) {
    /* 代码 */
  }
};
```

当一个rectangle替代品为了满足新对象geometryApplication的getLargestRectangle 的时候，它仅仅需要rectangle的area()方法，但它却违反了LSP（因为他根本用不到其中drawRectangles方法才能用到的draw方法）。

静态耦合

静态类型语言里的另外一个导致违反ISP的原因是静态耦合，在静态类型语言里，接口在一个松耦合设计程序里扮演了重大角色。不管是在动态语言还是在静态语言，有时候一个对象都可能需要多个客户端用户进行通信（比如共享状态），对静态类型语言，最好的解决方案是使用[Role Interfaces](#)，它允许用户和该对象进行交互（而该对象可能需要在多个角色）作为它的实现来对用户和无关的行为进行解耦。在JavaScript里就没有这种问题了，因为对象都被动态语言所特有的优点进行解耦了。

语义耦合

导致违反ISP的一个通用原因，动态语言和静态类型语言都有，那就是语义耦合，所谓语义耦合就是互相依赖，也就是一个对象的行为依赖于另外一个对象，那就意味着，如果一个用户改变了其中一个行为，很有可能会影响另外一个使用用户。这也违反单一职责原则了。可以通过继承和对象替代来解决这个问题。

可扩展性

另外一个导致问题的原因是关于可扩展性，很多人在举例的时候都会举关于callback的例子用来展示可扩展性（比如ajax里成功以后的回调设置）。如果想这样的接口需要一个实现并且这个实现的对象里有很多熟悉或方法的话，ISP就会变得很重要了，也就是说当一个接口interface变成了一个需求实现很多方法的时候，他的实现将会变得异常复杂，而且有可能导致这些接口承担一个没有粘性的职责，这就是我们经常提到的胖接口。

总结

JavaScript里的动态语言特性，使得我们实现非粘性接口的影响力比静态类型语言小，但接口隔离原则在JavaScript程序设计模式里依然有它发挥作用的地方。

(22) S.O.L.I.D五大原则之依赖倒置原则DIP

前言

本章我们要讲解的是S.O.L.I.D五大原则JavaScript语言实现的第5篇，依赖倒置原则LSP (The Dependency Inversion Principle)。

英文原文：<http://freshbrewedcode.com/derekgreer/2012/01/22/solid-javascript-the-dependency-inversion-principle/>

依赖倒置原则

依赖倒置原则的描述是：

A. High-level modules should not depend on low-level modules. Both should depend on abstractions.

高层模块不应该依赖于低层模块，二者都应该依赖于抽象

B. Abstractions should not depend upon details. Details should depend upon abstractions.

抽象不应该依赖于细节，细节应该依赖于抽象

依赖倒置原则的最重要问题就是确保应用程序或框架的主要组件从非重要的底层组件实现细节解耦出来，这将确保程序的最重要的部分不会因为低层次组件的变化修改而受影响。

该原则的第一部分是关于高层模块和低层模块之间的耦合方式，在传统的分成架构中，高层模块（封装了程序的核心业务逻辑）总依赖于低层的一些模块（一些基础点）。当应用依赖倒置原则的时候，关系就反过来了。和高层模块依赖于低层模块不同，依赖倒置是让低层模块依赖于高层模块里定义的接口。举例来说，如果要给程序进行数据持久化，传统的设计是核心模块依赖于一个持久化模块的API，而根据依赖倒置原则重构以后，则是核心模块需要定义持久化的API接口，然后持久化的实现实例需要实现核心模块定义的这个API接口。

该原则的第二部分描述的是抽象和细节之间的正确关系。理解这一部分，通过了解C++语言比较有帮助，因为他的适用性比较明显。

不像一些静态类型的语言，C++没有提供一个语言级别的概念来定义接口，那类定义和类实现之间到底是怎么样的呢，在C++里，类通过头文件的形式来定义，其中定义了源文件需要

实现的类成员方法和变量。因为所有的变量和私有方法都定义在头文件里，所以可以用来抽象以便和实现细节之前解耦出来。通过定义只定义抽象方法来实现（C++里是抽象基类）接口这个概念用于实现类来实现。

DIP and JavaScript

因为JavaScript是动态语言，所以不需要去为了解耦而抽象。所以抽象不应依赖于细节这个改变在JavaScript里没有太大的影响，但高层模块不应依赖于低层模块却有很大的影响。

在当静态类型语言的上下文里讨论依赖倒置原则的时候，耦合的概念包括语义（semantic）和物理（physical）两种。这就是说，如果一个高层模块依赖于一个低层模块，也就是不仅耦合了语义接口，也耦合了在底层模块里定义的物理接口。也就是说高层模块不仅要从第三方类库解耦出来，也需要从原生的低层模块里解耦出来。

为了解释这一点，想象一个.NET程序可能包含一个非常有用的高层模块，而该模块依赖于一个低层的持久化模块。当作者需要在持久化API里增加一个类似的接口的时候，不管依赖倒置原则有没有使用，高层模块在不重新实现这个低层模块的新接口之前是没有办法在其它的程序里得到重用的。

在JavaScript里，依赖倒置原则的适用性仅仅限于高层模块和低层模块之间的语义耦合，比如，DIP可以根据需要去增加接口而不是耦合低层模块定义的隐式接口。

为了来理解这个，我们看一下如下例子：

```
$.fn.trackMap = function(options) {
    var defaults = {
        /* defaults */
    };
    options = $.extend({}, defaults, options);

    var mapOptions = {
        center: new google.maps.LatLng(options.latitude, options.longitude
    ),
        zoom: 12,
        mapTypeId: google.maps.MapTypeId.ROADMAP
    },
    map = new google.maps.Map(this[0], mapOptions),
    pos = new google.maps.LatLng(options.latitude, options.longitude);

    var marker = new google.maps.Marker({
        position: pos,
        title: options.title,
        icon: options.icon
    });

    marker.setMap(map);
```

```

    options.feed.update(function(latitude, longitude) {
        marker.setMap(null);
        var newLatLng = new google.maps.LatLng(latitude, longitude);
        marker.position = newLatLng;
        marker.setMap(map);
        map.setCenter(newLatLng);
    });

    return this;
};

var updater = (function() {
    // private properties

    return {
        update: function(callback) {
            updateMap = callback;
        }
    };
})();

$("#map_canvas").trackMap({
    latitude: 35.044640193770725,
    longitude: -89.98193264007568,
    icon: 'http://bit.ly/zjnGDe',
    title: 'Tracking Number: 12345',
    feed: updater
});

```

在上述代码里，有个小型的JS类库将一个DIV转化成Map以便显示当前跟踪的位置信息。trackMap函数有2个依赖：第三方的Google Maps API和Location feed。该feed对象的职责是当icon位置更新的时候调用一个callback回调（在初始化的时候提供的）并且传入纬度latitude和精度longitude。Google Maps API是用来渲染界面的。

feed对象的接口可能按照装，也可能没有照装trackMap函数的要求去设计，事实上，他的角色很简单，着重在简单的不同实现，不需要和Google Maps这么依赖。介于trackMap语义上耦合了Google Maps API，如果需要切换不同的地图提供商的话那就不得不对trackMap函数进行重写以便可以适配不同的provider。

为了将Google maps类库的语义耦合翻转过来，我们需要重写设计trackMap函数，以便对一个隐式接口（抽象出地图提供商provider的接口）进行语义耦合，我们还需要一个适配Google Maps API的一个实现对象，如下是重构后的trackMap函数：

```

$.fn.trackMap = function(options) {
    var defaults = {

```

```

        /* defaults */
    };

    options = $.extend({}, defaults, options);

    options.provider.showMap(
        this[0],
        options.latitude,
        options.longitude,
        options.icon,
        options.title);

    options.feed.update(function(latitude, longitude) {
        options.provider.updateMap(latitude, longitude);
    });

    return this;
};

$("#map_canvas").trackMap({
    latitude: 35.044640193770725,
    longitude: -89.98193264007568,
    icon: 'http://bit.ly/zjnGDe',
    title: 'Tracking Number: 12345',
    feed: updater,
    provider: trackMap.googleMapsProvider
});

```

在该版本里，我们重新设计了trackMap函数以及需要的一个地图提供商接口，然后将实现的细节挪到了一个单独的googleMapsProvider组件，该组件可能独立封装成一个单独的JavaScript模块。如下是我的googleMapsProvider实现：

```

trackMap.googleMapsProvider = (function() {
    var marker, map;

    return {
        showMap: function(element, latitude, longitude, icon, title) {
            var mapOptions = {
                center: new google.maps.LatLng(latitude, longitude),
                zoom: 12,
                mapTypeId: google.maps.MapTypeId.ROADMAP
            },
            pos = new google.maps.LatLng(latitude, longitude);

            map = new google.maps.Map(element, mapOptions);

            marker = new google.maps.Marker({
                position: pos,
                title: title,
                icon: icon
            });
        }
    };
});

```

```

        marker.setMap(map);
    },
    updateMap: function(latitude, longitude) {
        marker.setMap(null);
        var newLatLng = new google.maps.LatLng(latitude, longitude);
        marker.position = newLatLng;
        marker.setMap(map);
        map.setCenter(newLatLng);
    }
};
})();

```

做了上述这些改变以后，trackMap函数将变得非常有弹性了，不必依赖于Google Maps API，相反可以任意替换其它的地图提供商，那就是说可以按照程序的需求去适配任何地图提供商。

何时依赖注入？

有点不太相关，其实依赖注入的概念经常和依赖倒置原则混在一起，为了澄清这个不同，我们有必要来解释一下：

依赖注入是控制反转的一个特殊形式，反转的意思一个组件如何获取它的依赖。依赖注入的意思就是：依赖提供给组件，而不是组件去获取依赖，意思是创建一个依赖的实例，通过工厂去请求这个依赖，通过Service Locator或组件自身的初始化去请求这个依赖。依赖倒置原则和依赖注入都是关注依赖，并且都是用于反转。不过，依赖倒置原则没有关注组件如何获取依赖，而是只关注高层模块如何从低层模块里解耦出来。某种意义上说，依赖倒置原则是控制反转的另外一种形式，这里反转的是哪个模块定义接口（从低层里定义，反转到高层里定义）。

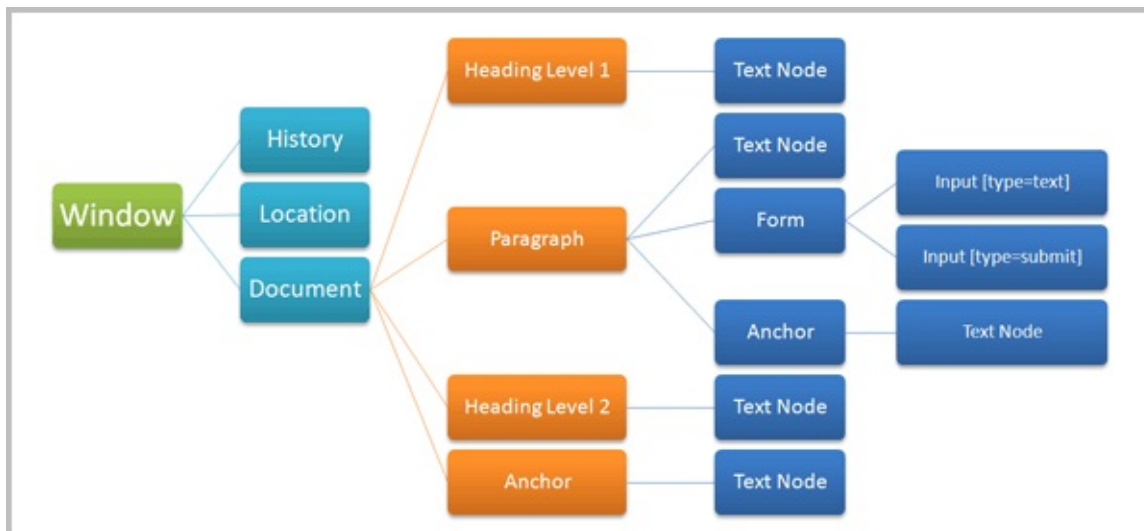
总结

这是五大原则的最后一篇了，在这5篇文字里我们看到了SOLID如何在JavaScript里实现的，不同的原则在JavaScript里通过不同的角度来说明的。（大叔注：其实大叔觉得虽然有点不伦不类，但从另外一个层面上说，大体的原则在各种语言上其实还是一样的。）

(23) JavaScript与DOM (上) ——也适用于新手

文档对象模型Document Object Model

DOM (Document Object Model , 文档对象模型) 是一个通过和JavaScript进行内容交互的API。Javascript和DOM一般经常作为一个整体，因为Javascript通常都是用来进行DOM操作和交互的。



主要内容来自：<http://net.tutsplus.com/tutorials/javascript-ajax/javascript-and-the-dom-series-lesson-1/>

关于DOM，有些知识需要注意：

1. window对象作为全局对象，也就是说你可以通过window来访问全局对象。
 1. 属性在对象下面以变量的形式存放，在页面上创建的所有全局对象都会变成window对象的属性。
 2. 方法在对象下面以函数的形式存放，因为左右的函数都存放在window对象下面，所以他们也可以称为方法。
2. DOM为web文档创建带有层级的结果，这些层级是通过node节点组成，这里有几种DOM node类型，最重要的是Element, Text, Document。
 1. Element节点在页面里展示的是一个元素，所以如果你有段落元素()，你可以通过这个DOM节点来访问。
 2. Text节点在页面里展示的所有文本相关的元素，所以如果你的段落有文本在里面的话，

你可以直接通过DOM的Text节点来访问这个文本

3. Document节点代表是整个文档，它是DOM的根节点。

3. 每个引擎对DOM标准的实现有一些轻微的不同。例如，Firefox浏览器使用的Gecko引擎有着很好的实现（尽管没有完全遵守W3C规范），但IE浏览器使用的Trident引擎的实现却不完整而且还有bug，给开发人言带来了很多问题。

如果你正在使用Firefox，我推荐你立即下载Firebug插件，对于你了解DOM结构非常有用。

Web上的JavaScript Script元素

当你在网站页面上使用JavaScript的时候，需要使用SCRIPT元素：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
  <html xmlns="http://www.w3.org/1999/xhtml" lang="en">
    <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
      <title>JavaScript!</title>
    </head>
    <body>
      <script type="text/javascript">
        // <![CDATA[
          // ]]>
        </script>
      </body>
    </html>
```

上述代码，严格来说SCRIPT的TYPE属性应该设置为application/javascript，但是由于IE不支持这个，所以平时我们不得不写成text/javascript或者直接去掉type。另外你也可以看到在SCRIPT元素里的注释行//，浏览器就不会再解析成XHTML标签了。

Defer属性

任何在SCRIPT元素里声明的代码在页面加载的时候都会运行，唯一一个例外是给SCRIPT元素加上一个defer属性。defer属性告诉浏览器加载完HTML文档以后再执行JS代码，但这个属性只能在IE下使用。

连接外部脚本

如果你了解外部脚本，只需要简单地在SCRIPT上使用SRC属性就行了，使用单独的JS文件的好处是可以缓存，而且也不需要担心CDATA方面的问题：

```
<script type="text/javascript" src="my-script.js"></script>
```

JavaScript必备

在我们继续DOM之前，我们来复习一下JavaScript的核心必备知识，如果你还不了解，也没关系，我们在这一章节将稍微花点时间来回顾一下。

JavaScript有几种数据类型：Number, String, Boolean, Object, Undefined and Null。

单行注释使用双斜杠//，双斜杠后面的所有文字都会被注释掉，多行注意使用/和/括住。

Number

在JavaScript里所有的Number都是浮点型的，当声明一个数字变量的时候，记得不要使用任何引号。

```
// 注：使用var类声明变量
var leftSide = 100;
var topSide = 50;
var areaOfRectangle = leftSide * topSide; // = 5000
```

String

JavaScript里声明字符串特别简单，和其它语言一样，在JS里使用单引号或双引号都可以。

```
var firstPart = 'Hello';
var secondPart = 'World!';
var allOfIt = firstPart + ' ' + secondPart; // Hello World!
// +符合是字符连接符。也用于数字相加
```

Boolean

布尔类型用于条件判断，布尔类型是只有2个值：true和false。任何使用逻辑操作符的比较都会返回布尔值。

```
5 === (3 + 2); // = true
// 你也可以将布尔值赋给一个变量
var veryTired = true;
// 这样使用
if (veryTired) {
    // 执行代码
}
```

===也是比较操作符，不仅比较数值，还比较类型。

Function

函数是特殊的对象。

```
// 使用function操作符来声明新函数
function myFunctionName(arg1, arg2) {
    // 函数代码
}

// 你也可以声明匿名函数
function (arg1, arg2) {
    // Function code goes here.
}

// 运行函数很简单，直接在函数名称后面加上小括号就可以了
// 或者也可以带上参数
myFunctionName(); // 无参
myFunctionName('foo', 'bar'); // 有参数

// 也可以使用自调用
(function () {
    // 这里自调用函数
})();
```

Array

数组也是特殊的对象，它包含了一批值（或对象），访问这些数据的话需要使用数字索引：

```
// 2种方式声明数组

// 字面量：
var fruit = ['apple', 'lemon', 'banana'];

// Array构造函数：
var fruit = new Array('apple', 'lemon', 'banana');

fruit[0]; // 访问第1个项(apple)
fruit[1]; // 访问第2个项(lemon)
fruit[2]; // 访问第3个项(banana)
```

Object

一个对象是一个key-value的集合，和数组相似，唯一的不同是你可以为每个数据定义一个名称。

```
// 2种类型定义Object对象

// 字面量（大括号）
var profile = {
```



```
    name: 'Bob',
    age: 99,
    job: 'Freelance Hitman'
};

// 使用Object构造函数
var profile = new Object();
profile.name = 'Bob';
profile.age = 99;
profile.job = 'Freelance Hitman';
```

IF/Else语句

JS里使用最多的语句莫过于条件语句了：

```
var legalDrinkingAge = 21;
var yourAge = 29;

if ( yourAge >= legalDrinkingAge ) {
    alert('You can drink.');
```

```
} else {
    alert('Sorry, you cannot drink.');
```

JavaScript操作符

建议你访问[这个页面](#)来查看所有的JS操作符，这里我仅仅给出一些例子：

```
// 加减乘除
var someMaths = 2 + 3 + 4 - 10 * 100 / 2;

// 等于
if ( 2 == (5 - 3) ) { /* 代码 */ } // == 比较是否相等

// 不等于
if ( 2 != (5 - 3) ) { /* 代码 */ }

// 严格等于（推荐）
2 === 2 // 代替 2 == 2
2 !== 3 // 代替 2 != 3

// 赋值：
var numberOfFruit = 9;
numberOfFruit -= 2; // 等价于 "numberOfFruit = numberOfFruit - 2"
numberOfFruit += 2; // 等价于 "numberOfFruit = numberOfFruit + 2"
```

Loop循环

Loop循环在是遍历数组或者对象的所有成员的时候非常方便，JavaScript里使用最多的是FOR和WHILE语句。

```

var envatoTutSites = ['NETTUTS', 'PSDTUTS', 'AUDIOTUTS', 'AETUTS', 'VECTO
RTUTS'];

// WHILE循环
var counter = 0;
var lengthOfArray = envatoTutSites.length;
while (counter < lengthOfArray) {
    alert(envatoTutSites[counter]);
    counter++; // 等价于counter += 1;
}

// FOR循环
// i只是用于迭代, 可以任意取名
for (var i = 0, length = envatoTutSites.length; i < length; i++) {
    alert(envatoTutSites[i]);
}

```

DOM正文

访问DOM节点

我们来个例子, 一个HTML里包含一段文本和一个无序的列表。

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.o
rg/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en">
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=U
TF-8" />
        <title>JavaScript!</title>
    </head>
    <body>

        <p id="intro">My first paragraph...</p>

        <ul>
            <li>List item 1</li>
            <li>List item 1</li>
            <li>List item 1</li>
            <li>List item 1</li>
            <li>List item 1</li>
        </ul>

        <script type="text/javascript">
            // <![CDATA[

                // ]]>
        </script>

    </body>
</html>

```

上面例子里, 我们使用getElementById DOM方法来访问p段落, 在SCRIPT里添加如下代

本文档使用 [看云](#) 构建

码：

```
var introParagraph = document.getElementById('intro');
// 现在有了该DOM节点，这个DOM节点展示的是该信息段落
```

变量introParagraph现在已经引用到该DOM节点上了，我们可以对该节点做很多事情，比如查询内容和属性，或者其它任何操作，甚至可以删除它，克隆它，或者将它移到DOM树的其它节点上。

文档上的任何内容，我们都可以使用JavaScript和DOM API来访问，所以类似地，我们也可以访问上面的无序列表，唯一的问题是元素没有ID属性，如果ID的话就可以使用相同的方式，或者使用如下getElementsByTagName方式：

```
var allUnorderedLists = document.getElementsByTagName('ul');
// 'getElementsByTagName' 返回的是一个节点集合
// - 和数组有点相似
```

getElementsByTagName

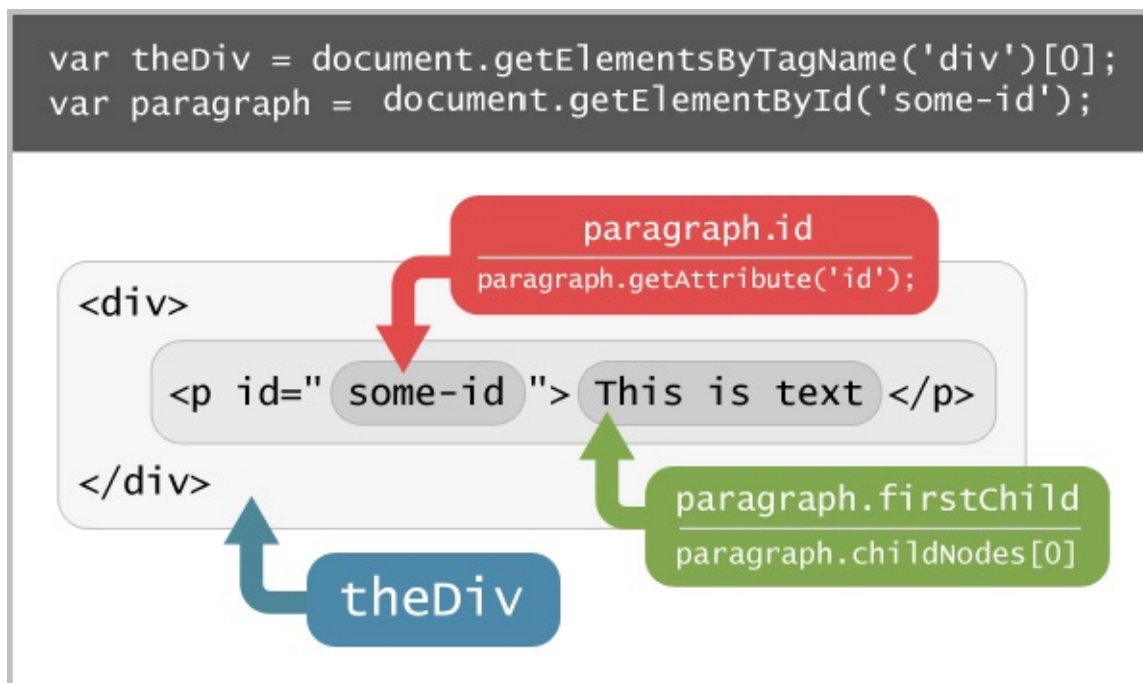
getElementsByTagName方法返回的是一个节点集合，和数组类似也有length属性，重要的一个特性是它是live的——如果你在该元素里添加一个新的li元素，这个集合就会自动更新，介于他和数组类型，所以可以和访问数组一样的方法来访问，所以从0开始：

```
// 访问无序列表：[0]索引
var unorderedList = document.getElementsByTagName('ul')[0];

// 获取所有的li集合：
var allListItems = unorderedList.getElementsByTagName('li');

// 循环遍历
for (var i = 0, length = allListItems.length; i < length; i++) {
    // 弹出该节点的text内容
    alert(allListItems[i].firstChild.data);
}
```

以下图例更清晰地展示了DOM获取的知识：



DOM穿梭

“穿梭”这个词主要是用来描述通过DOM查找节点，DOM API提供了大量的节点属性让我们来往上或者往下查询节点。

所有的节点都有这些属性，都是可以用于访问相关的node节点：

1. `Node.childNodes`: 访问一个单元素下所有的直接子节点元素，可以是一个可循环的类数组对象。该节点集合可以保护不同的类型的子节点（比如text节点或其他元素节点）。
2. `Node.firstChild`: 与 ‘childNodes’ 数组的第一个项(‘Element.childNodes[0] ’)是同样的效果，仅仅是快捷方式。
3. `Node.lastChild`: 与 ‘childNodes’ 数组的最后一个项 (‘Element.childNodes[Element.childNodes.length-1] ’)是同样的效果，仅仅是快捷方式。shortcut.
4. `Node.parentNode`: 访问当前节点的父节点，父节点只能有一个，祖节点可以用 ‘Node.parentNode.parentNode’ 的形式来访问。
5. `Node.nextSibling`: 访问DOM树上与当前节点同级别的下一个节点。
6. `Node.previousSibling`: 访问DOM树上与当前节点同级别的上一个节点。



通过这张图，理解起来就简单多了，但有个非常重要的知识点：那就是元素之间不能有空格，如果ul和li之间有空格的话，就会被认为是内容为空的text node节点，这样ul.childNodes[0]就不是第一个li元素了。相应地，的下一个节点也不是，因为和之间有一个空行的节点，一般遇到这种情况需要遍历所有的子节点然后判断nodeType类型，1是元素，2是属性，3是text节点，详细的type类型可以通过[此地址](#)：

```
Node.ELEMENT_NODE == 1
Node.ATTRIBUTE_NODE == 2
Node.TEXT_NODE == 3
Node.CDATA_SECTION_NODE == 4
Node.ENTITY_REFERENCE_NODE == 5
Node.ENTITY_NODE == 6
Node.PROCESSING_INSTRUCTION_NODE == 7
Node.COMMENT_NODE == 8
Node.DOCUMENT_NODE == 9
Node.DOCUMENT_TYPE_NODE == 10
Node.DOCUMENT_FRAGMENT_NODE == 11
Node.NOTATION_NODE == 12
```

总结

原生的DOM方法和属性足够我们日常的应用了，本章节我们只列举了一些例子，下一章节我们列举更多的例子，还会包括浏览器事件模型。

(24) JavaScript与DOM (下)

介绍

上一章我们介绍了JavaScript的基本内容和DOM对象的各个方面，包括如何访问node节点。本章我们将讲解如何通过DOM操作元素并且讨论浏览器事件模型。

本文参考：<http://net.tutsplus.com/tutorials/javascript-ajax/javascript-and-the-dom-lesson-2/>

操作元素

上一章节我们提到了DOM节点集合或单个节点的访问步骤，每个DOM节点都包括一个属性集合，大多数的属性都为相应的功能提供了抽象。例如，如果有一个带有ID属性intro的文本元素，你可以很容易地通过DOM API来改变该元素的颜色：

```
document.getElementById('intro').style.color = '#FF0000';
```

为了理解这个API的功能，我们一步一步分开来看就非常容易理解了：

```
var myDocument = document;
var myIntro = myDocument.getElementById('intro');
var myIntroStyles = myIntro.style;

// 现在，我们可以设置颜色了：
myIntroStyles.color = '#FF0000';
```

现在，我们有了该文本的style对象的引用了，所以我们可以添加其它的CSS样式：

```
myIntroStyles.padding = '2px 3px 0 3px';
myIntroStyles.backgroundColor = '#FFF';
myIntroStyles.marginTop = '20px';
```

这里我们只是要了基本的CSS属性名称，唯一区别是CSS属性的名称如果带有-的话，就需要去除，比如用marginTop代替margin-top。例如，下面的代码是不工作的，并且会抛出语法错误：

```
myIntroStyles.padding-top = '10em';
```

```
// 产生语法错误：
```

本文档使用 [看云](#) 构建

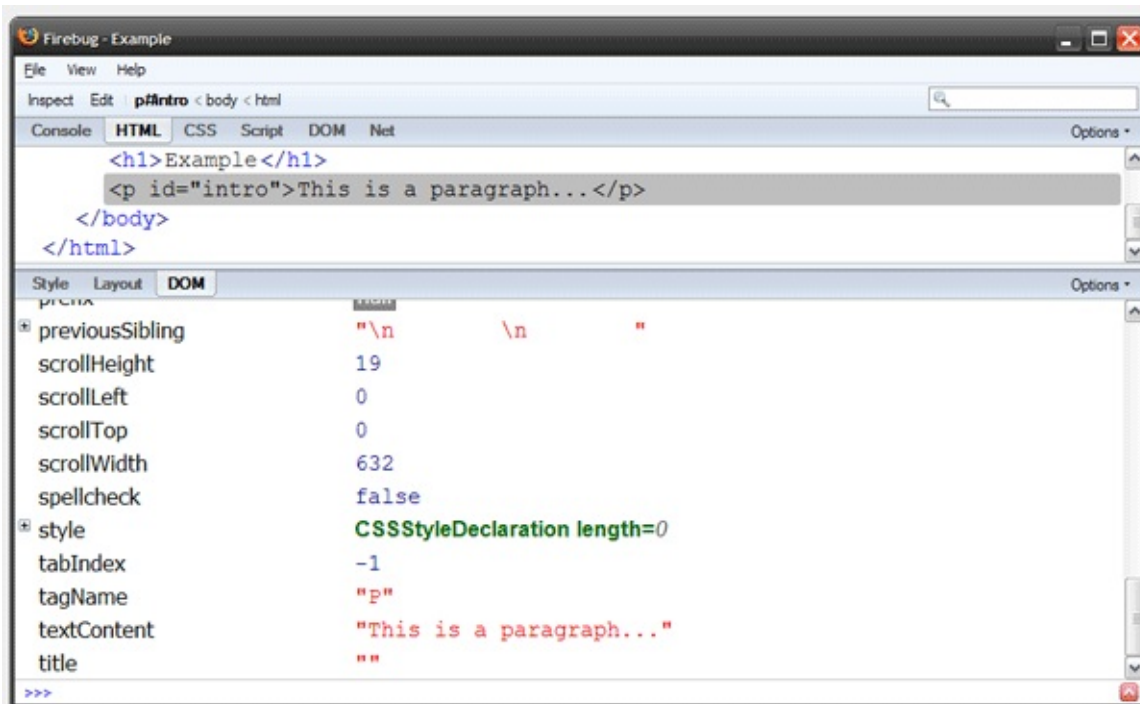
```
// 在JavaScript里横线-是减法操作符
// 而且也没有这样的属性名称
```

属性可以像数组一样访问，所以利用这个知识我们可以创建一个函数来改变任何给定元素的样式：

```
function changeStyle(elem, property, val) {
    elem.style[property] = val; // 使用[]来访问属性
}

// 使用上述的函数：
var myIntro = document.getElementById('intro'); // 获取intro文本对象
changeStyle(myIntro, 'color', 'red');
```

这仅仅是个例子，所以该函数也许没什么用，语法上来说，直接用还是会快点，例如（`elem.style.color = 'red'`）。除了style属性以外，一个节点（或元素）也还有其他很多属性可以操作，如果你使用Firebug，点击DOM选项卡可以看到所有该节点（或元素）的所有属性：



所有的属性都可以通过点标示符来访问（例如：`Element.tabIndex`）。不是所有的属性都是原始数据类型（strings, numbers, Booleans等等），style属性也是一个包含自己属性的对象，很多元素的属性都是只读的，也就是说不能修改他们的值。例如，你不能直接修改一个节点的parentNode属性，如果你修改只读属性的时候浏览器会抛出错误：例如，抛出错

误 “setting a property that has only a getter” ，只是我们需要注意的。

通常DOM操作都是改变原始的内容，这里有几种方式来实现这个，最简单的是使用innerHTML属性，例如：

```
var myIntro = document.getElementById('intro');

// 替换当前的内容
myIntro.innerHTML = 'New content for the <strong>amazing</strong> paragraph!';

// 添加内容到当前的内容里
myIntro.innerHTML += '... some more content...';
```

唯一的问题是该方法没在规范里定义，而且在DOM规范里也没有定义，如果你不反感的话请继续使用，因为它比我们下面要讨论其它的方法快多了。

Node节点

通过DOM API创建内容的时候需要注意node节点的2种类型，一种是元素节点，一种是text节点，上一章节已经列出了所有的节点类型，这两种需要我们现在特别注意。创建元素可以通过createElement方法，而创建text节点可以使用createTextNode，相应代码如下：

```
var myIntro = document.getElementById('intro');

// 添加内容
var someText = 'This is the text I want to add';
var textNode = document.createTextNode(someText);
myIntro.appendChild(textNode);
```

这里我们使用了appendChild方法将新text节点附件到文本字段，这样做比非标准的innerHTML方法显得有点长，但了解这些原理依然很重要，这里有一个使用DOM方法的更详细例子：

```
var myIntro = document.getElementById('intro');

// 添加新连接到文本节点
// 首先，创建新连接元素
var myNewLink = document.createElement('a'); //
myNewLink.href = 'http://google.com'; //
myNewLink.appendChild(document.createTextNode('Visit Google'));
// Visit Google

// 将内容附件到文本节点
myIntro.appendChild(myNewLink);
```

另外DOM里还有一个insertBefore方法用于在节点前面插入内容，通过insertBefore和appendChild我们可以实现自己的insertAfter函数：

```
// 'Target'是DOM里已经存在的元素
// 'Bullet'是要插入的新元素

function insertAfter(target, bullet) {
    target.nextSibling ?
        target.parentNode.insertBefore(bullet, target.nextSibling)
        : target.parentNode.appendChild(bullet);
}

// 使用了3目表达式：
// 格式：条件?条件为true时的表达式：条件为false时的表达式
```

上面的函数首先检查target元素的同级下一个节点是否存在，如果存在就在该节点前面添加bullet节点，如果不存在，就说明target是最后一个节点了，直接在后面append新节点就可以了。DOM API没有提供insertAfter是因为真的没必要了——我们可以自己创建。

DOM操作有很多内容，上面你看到的只是其中一部分。

Event事件

浏览器事件是所有web程序的核心，通过这些事件我们定义将要发生的行为，如果在页面里有个按钮，那点击此按钮之前你需要验证表单是否合法，这时候就可以使用click事件，下面列出的最标准的事件列表：

注：正如我们上章所说的，DOM和JavaScript语言是2个单独的东西，浏览器事件是DOM API的一部分，而不是JavaScript的一部分。

鼠标事件

1. 'mousedown' – 鼠标设备按下一个元素的时候触发mousedown事件。
2. 'mouseup' – 鼠标设备从按下的元素上弹起的时候触发mouseup事件。
3. 'click' – 鼠标点击元素的时候触发click事件。
4. 'dblclick' – 鼠标双击元素的时候触发dblclick事件。
5. 'mouseover' – 鼠标移动到某元素上的时候触发mouseover事件。
6. 'mouseout' – 鼠标从某元素离开的时候触发mouseout事件。
7. 'mousemove' – 鼠标在某元素上移动但未离开的时候触发mousemove事件。

键盘事件

1. 'keypress' – 按键按下的时候触发该事件。

2. 'keydown' – 按键按下的时候触发该事件，并且在keypress事件之前。
3. 'keyup' – 按键松开的时候触发该事件，在keydown和keypress事件之后。

表单事件

1. 'select' – 文本字段 (input, textarea等) 的文本被选择的时候触发该事件。
2. 'change' – 控件失去input焦点的时候触发该事件 (或者值被改变的时候)。
3. 'submit' – 表单提交的时候触发该事件。
4. 'reset' – 表单重置的时候触发该事件。
5. 'focus' – 元素获得焦点的时候触发该事件，通常来自鼠标设备或Tab导航。
6. 'blur' – 元素失去焦点的时候触发该事件，通常来自鼠标设备或Tab导航。

其它事件

1. 'load' – 页面加载完毕 (包括内容、图片、frame、object) 的时候触发该事件。
2. 'resize' – 页面大小改变的时候触发该事件 (例如浏览器缩放)。
3. 'scroll' – 页面滚动的时候触发该事件。
4. 'unload' – 从页面或frame删除所有内容的时候触发该事件 (例如离开一个页面)。

还有很多各种各样的事件，上面展示的事件是我们在JavaScript里最常用的事件，有些事件在跨浏览器方面可能有所不同。还有其它浏览器实现的一些属性事件，例如Gecko实现的DOMContentLoaded或DOMMouseScroll等，Gecko的详细事件列表请[查看这里](#)。

事件处理

我们将了事件，但是还没有将到如何将处理函数和事件管理起来，使用这些事件之前，你首先要注册这些事件句柄，然后描述该事件发生的时候该如何处理，下面的例子展示了一个基本的事件注册模型：

基本事件注册：

```
<!-- HTML -->
<button id="my-button">Click me!</button>
```

```
// JavaScript:
var myElement = document.getElementById('my-button');

// 事件处理句柄:
function buttonClick() {
    alert('You just clicked the button!');
}

// 注册事件
```

```
myElement.onclick = buttonClick;
```

使用document.getElementById命令，通过ID=my-button获取该button对象，然后创建一个处理函数，随后将该函数赋值给该DOM的onclick属性。就这么简单！

基本事件注册是非常简单的，在事件名称前面添加前缀on作为DOM的属性就可以使用了，这是事件处理的基本核心，但下面的代码我不推荐使用：

```
<button onclick="return buttonClick()">Click me!</button>
```

上述Inline的事件处理方式不利用页面维护，建议将这些处理函数都封装在单独的js文件，原因和CSS样式的一样的。

高级事件注册：

别被标题迷惑了，“高级”不意味着好用，实际上上面讨论的基本事件注册是我们大部分时候用的方式，但有一个限制：不能绑定多个处理函数到一个事件上。这也是我们要讲解该小节原因：

该模型运行你绑定多个处理句柄到一个事件上，也就是说一个事件触发的时候多个函数都可以执行，另外，该模型也可以让你很容易里删除某个已经绑定的句柄。

严格来说，有2中不同的模型：W3C模型和微软模型，除IE之外W3C模型支持所有的现代浏览器，而微软模型只支持IE，使用[W3C模型](#)的代码如下：

```
// 格式: target.addEventListener( type, function, useCapture );
// 例子:
var myIntro = document.getElementById('intro');
myIntro.addEventListener('click', introClick, false);
```

使用[IE模型](#)的代码如下：

```
// 格式: target.attachEvent ( 'on' + type, function );
// 例子:
var myIntro = document.getElementById('intro');
myIntro.attachEvent('onclick', introClick);
```

introClick的代码如下：

```
function introClick() {
    alert('You clicked the paragraph!');
}
```

事实上，要做出通用的话，我们可以自定义一个函数以支持跨浏览器：

```
function addEvent(elem, type, fn) {
    if (elem.attachEvent) {
        elem.attachEvent('on' + type, fn);
        return;
    }
    if (elem.addEventListener) {
        elem.addEventListener(type, fn, false);
    }
}
```

该函数首先检查attachEvent和addEventListener属性，谁可以就用谁，这两种类型的模型都支持删除句柄功能，参考下面的removeEvent函数。

```
function removeEvent(elem, type, fn) {
    if (elem.detachEvent) {
        elem.detachEvent('on' + type, fn);
        return;
    }
    if (elem.removeEventListener) {
        elem.removeEventListener(type, fn, false);
    }
}
```

你可以这样使用：

```
var myIntro = document.getElementById('intro');
addEvent(myIntro, 'click', function () {
    alert('YOU CLICKED ME!!!');
});
```

注意到我们传入了一个匿名函数作为第三个参数，JavaScript运行我们定义和执行匿名函数，这种匿名函数特别适合作为参数传递，实际上我们也可以传递有名的函数（代码如下），但是你们函数更容易做。

如果你只想在第一次click的时候触发一个函数，你可以这么做：

```
// 注意：前提是我们已经定好了addEvent/removeEvent函数
```

```
// (定义好了才能使用哦)

var myIntro = document.getElementById('intro');
addEvent(myIntro, 'click', oneClickOnly);

function oneClickOnly() {
    alert('WOW!');
    removeEvent(myIntro, 'click', oneClickOnly);
}
```

当第一次触发以后，我们就立即删除该句柄，但是有匿名函数的话却很难将自身的引用删除，不过实际上可以通过如下的形式来做（只不过有点麻烦）：

```
addEvent(myIntro, 'click', function () {
    alert('WOW!');
    removeEvent(myIntro, 'click', arguments.callee);
});
```

这里我们是有了arguments对象的callee属性，arguments对象包含了所有传递进来的参数以及该函数自身(callee)，这样我们就可以放心地删除自身的引用了。

关于W3C和微软模型还有其他的少许差异，比如this，在触发事件的时候函数中的this一般都是该元素上下文，，也就说this引用该元素自身，在基本事件注册和W3C模型中都没有问题，但在微软模型的实现里却可能出错，请参考如下代码：

```
function myEventHandler() {
    this.style.display = 'none';
}

// 正常工作，this是代表该元素
myIntro.onclick = myEventHandler;

// 正常工作，this是代表该元素
myIntro.addEventListener('click', myEventHandler, false);

// 不正常，这时候的this是代表Window对象
myIntro.attachEvent('onclick', myEventHandler);
```

这里有一些方式可以避免这个问题，最简单的方式是使用前面的基本事件注册方式，或者是再做一个通用的addEvent，通用代码请参考[John Resig](#)或[Dean Edward](#)的文章。

Event对象

另外一个非常重要的内容是Event对象，当事件发生的时候出发某个函数，该Event对象将自动在函数内可用，该对象包含了很多事件触发时候的信息，但IE却没有这么实现，而是自己

实现的，IE浏览器是通过全局对象window下的event属性来包含这些信息，虽然不是大问题，但我们也要注意一下，下面的代码是兼容性的：

```
function myEventHandler(e) {

    // 注意参数e
    // 该函数调用的时候e是event对象（W3C实现）

    // 兼容IE的代码
    e = e || window.event;

    // 现在e就可以兼容各种浏览器了

}

// 这里可以自由地绑定事件了
```

这里判断e对象（Event对象）是否存在我们使用了OR操作符：如果e不存在（为null, undefined, 0等）的时候，将window.event赋值给e，否则的话继续使用e。通过这方式很快就能在多浏览器里得到真正的Event对象，如果你不喜欢这种方式的话，你可以使用if语句来处理：

```
if (!e) {
    e = window.event;
} // 没有else语句，因为e在其它浏览器已经定义了
```

另外Event对象下的命令和属性都很有用，遗憾的是不能全兼容浏览器，例如当你想取消默认的行为的时候你可以使用Event对象里的preventDefault()方法，但IE里不得不使用对象的returnValue属性值来控制，兼容代码如下：

```
function myEventHandler(e) {
    e = e || window.event;
    // 防止默认行为
    if (e.preventDefault) {
        e.preventDefault();
    } else {
        e.returnValue = false;
    }
}
```

例如，当你点击一个连接的时候，默认行为是导航到href里定义的地址，但有时候你想禁用这个默认行为，通过returnValue和preventDefault就可以实现，Event对象里的很多属性在浏览器里都不兼容，所以很多时候需要处理这些兼容性代码。

注意：现在很多JS类库都已经封装好了e.preventDefault代码，也就是说在IE里可用了，但是原理上依然是使用returnValue来实现的。

事件冒泡

事件冒泡，就是事件触发的时候通过DOM向上冒泡，首先要知道不是所有的事件都有冒泡。事件在一个目标元素上触发的时候，该事件将触发——触发祖先节点元素，直到最顶层的元素：

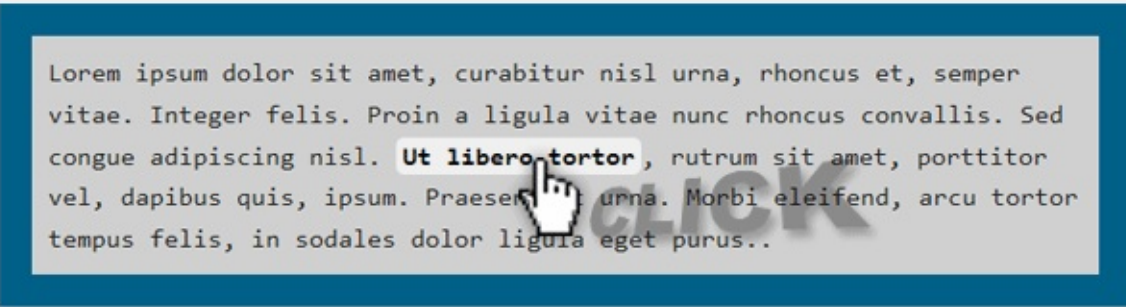


Diagram illustrating event bubbling. A hand cursor clicks on a link 'Ut libero tortor' within a paragraph. The event bubbles up from the link to the paragraph, then to the div, and finally to the body.

<pre> <div> <p>Lorem ipsum... Ut libero tortor </p> </div> </pre>	<p>In order of occurrence:</p> <ol style="list-style-type: none"> 1. <a> click event fires 2. <p> click event fires 3. <div> click event fires 4. <body> click event fires
---	--

如图所示，如果a连接被点击，触发触发连接的click事件，然后触发p的click事件，以此再触发div和body的click事件。顺序不变，而且不一定是在同时触发的。

这样你就可以利用该特性去处理自己的逻辑了，并且再任何时候都可以停止冒泡，比如，如果你只想冒泡到文本节点上，而不再进一步冒泡，你可以在p的click事件处理函数里停止冒泡：

```

function myParagraphEventHandler(e) {

    e = e || window.event;

    // 停止向上冒泡
    if (e.stopPropagation) {
        // W3C实现
        e.stopPropagation();
    } else {
        // IE实现
        e.cancelBubble = true;
    }

}

```



```
// 使用我们自定义的addEvent函数将myParagraphEventHandler绑定到click事件上：  
addEvent(document.getElementsByTagName('p')[0], 'click', myParagraphEvent  
Handler);
```

事件委托

举例来说，如果你有一个很多行的大表格，在每个上绑定点击事件是个非常危险的想法，因为性能是个大问题。流行的做法是使用事件委托。事件委托描述的是将事件绑定在容器元素上，然后通过判断点击的target子元素的类型来触发相应的事件。

```
var myTable = document.getElementById('my-table');  
  
myTable.onclick = function () {  
    // 处理浏览器兼容  
    e = e || window.event;  
    var targetNode = e.target || e.srcElement;  
  
    // 测试如果点击的是TR就触发  
    if (targetNode.nodeName.toLowerCase() === 'tr') {  
        alert('You clicked a table row!');  
    }  
}
```

事件委托依赖于事件冒泡，如果事件冒泡到table之前被禁用的话，那上面的代码就无法工作了。

总结

本章我们覆盖到了DOM元素的操作以及相关的浏览器事件模型，希望大家能对DOM有了进一步的了解。有任何问题，请留言讨论。

(25) 设计模式之单例模式

介绍

从本章开始，我们会逐步介绍在JavaScript里使用的各种设计模式实现，在这里我不会过多地介绍模式本身的理论，而只会关注实现。OK，正式开始。

在传统开发工程师眼里，单例就是保证一个类只有一个实例，实现的方法一般是先判断实例存在与否，如果存在直接返回，如果不存在就创建了再返回，这就确保了一个类只有一个实例对象。在JavaScript里，单例作为一个命名空间提供者，从全局命名空间里提供一个唯一的访问点来访问该对象。

正文

在JavaScript里，实现单例的方式有很多种，其中最简单的一个方式是使用对象字面量的方法，其字面量里可以包含大量的属性和方法：

```
var mySingleton = {
  property1: "something",
  property2: "something else",
  method1: function () {
    console.log('hello world');
  }
};
```

如果以后要扩展该对象，你可以添加自己的私有成员和方法，然后使用闭包在其内部封装这些变量和函数声明。只暴露你想暴露的public成员和方法，样例代码如下：

```
var mySingleton = function () {

  /* 这里声明私有变量和方法 */
  var privateVariable = 'something private';
  function showPrivate() {
    console.log(privateVariable);
  }

  /* 公有变量和方法（可以访问私有变量和方法） */
  return {
    publicMethod: function () {
      showPrivate();
    },
    publicVar: 'the public can see this!'
  };
};
```

```
var single = mySingleton();
single.publicMethod(); // 输出 'something private'
console.log(single.publicVar); // 输出 'the public can see this!'
```

上面的代码很不错了，但如果我们想做到只有在使用的时候才初始化，那该如何做呢？为了节约资源的目的，我们可以另外一个构造函数里来初始化这些代码，如下：

```
var Singleton = (function () {
    var instantiated;
    function init() {
        /*这里定义单例代码*/
        return {
            publicMethod: function () {
                console.log('hello world');
            },
            publicProperty: 'test'
        };
    }

    return {
        getInstance: function () {
            if (!instantiated) {
                instantiated = init();
            }
            return instantiated;
        }
    };
})();

/*调用公有的方法来获取实例:*/
Singleton.getInstance().publicMethod();
```

知道了单例如何实现了，但单例用在什么样的场景比较好呢？其实单例一般是用在系统间各种模式的通信协调上，下面的代码是一个单例的最佳实践：

```
var SingletonTester = (function () {

    //参数：传递给单例的一个参数集合
    function Singleton(args) {

        //设置args变量为接收的参数或者为空（如果没有提供的话）
        var args = args || {};
        //设置name参数
        this.name = 'SingletonTester';
        //设置pointX的值
        this.pointX = args.pointX || 6; //从接收的参数里获取，或者设置为默认值
        //设置pointY的值
        this.pointY = args.pointY || 10;

    }

})
```

```

//实例容器
var instance;

var _static = {
    name: 'SingletonTester',

    //获取实例的方法
    //返回Singleton的实例
    getInstance: function (args) {
        if (instance === undefined) {
            instance = new Singleton(args);
        }
        return instance;
    }
};
return _static;
})();

var singletonTest = SingletonTester.getInstance({ pointX: 5 });
console.log(singletonTest.pointX); // 输出 5

```

其它实现方式

方法1：

```

function Universe() {

    // 判断是否存在实例
    if (typeof Universe.instance === 'object') {
        return Universe.instance;
    }

    // 其它内容
    this.start_time = 0;
    this.bang = "Big";

    // 缓存
    Universe.instance = this;

    // 隐式返回this
}

// 测试
var uni = new Universe();
var uni2 = new Universe();
console.log(uni === uni2); // true

```

方法2：

```

function 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;
    };
}

// 测试
var uni = new Universe();
var uni2 = new Universe();
uni.bang = "123";
console.log(uni === uni2); // true
console.log(uni2.bang); // 123

```

方法3：

```

function Universe() {

    // 缓存实例
    var instance;

    // 重新构造函数
    Universe = function Universe() {
        return instance;
    };

    // 后期处理原型属性
    Universe.prototype = this;

    // 实例
    instance = new Universe();

    // 重设构造函数指针
    instance.constructor = Universe;

    // 其它功能
    instance.start_time = 0;
    instance.bang = "Big";

    return instance;
}

// 测试
var uni = new Universe();
var uni2 = new Universe();
console.log(uni === uni2); // true

// 添加原型属性
Universe.prototype.nothing = true;

```

```
var uni = new Universe();

Universe.prototype.everything = true;

var uni2 = new Universe();

console.log(uni.nothing); // true
console.log(uni2.nothing); // true
console.log(uni.everything); // true
console.log(uni2.everything); // true
console.log(uni.constructor === Universe); // true
```

方式4:

```
var Universe;

(function () {
    var instance;

    Universe = function Universe() {
        if (instance) {
            return instance;
        }

        instance = this;

        // 其它内容
        this.start_time = 0;
        this.bang = "Big";
    };
} ());

//测试代码
var a = new Universe();
var b = new Universe();
alert(a === b); // true
a.bang = "123";
alert(b.bang); // 123
```

参考资料

<https://github.com/shichuan/javascript-patterns/blob/master/design-patterns/singleton.html>

<http://www.addyosmani.com/resources/essentialjsdesignpatterns/book/#singletonpatternjavascript>

(26) 设计模式之构造函数模式

介绍

构造函数大家都很熟悉了，不过如果你是新手，还是有必要来了解一下什么叫构造函数的。构造函数用于创建特定类型的对象——不仅声明了使用的对象，构造函数还可以接受参数以便第一次创建对象的时候设置对象的成员值。你可以自定义自己的构造函数，然后在里面声明自定义类型对象的属性或方法。

基本用法

在JavaScript里，构造函数通常是认为用来实现实例的，JavaScript没有类的概念，但是有特殊的构造函数。通过new关键字来调用定义的正早函数，你可以告诉JavaScript你要创建一个新对象并且新对象的成员声明都是构造函数里定义的。在构造函数内部，this关键字引用的是新创建的对象。基本用法如下：

```
function Car(model, year, miles) {
    this.model = model;
    this.year = year;
    this.miles = miles;
    this.output= function () {
        return this.model + "走了" + this.miles + "公里";
    };
}

var tom= new Car("大叔", 2009, 20000);
var dudu= new Car("Dudu", 2010, 5000);

console.log(tom.output());
console.log(dudu.output());
```

上面的例子是个非常简单的构造函数模式，但是有点小问题。首先是使用继承很麻烦了，其次output()在每次创建对象的时候都重新定义了，最好的方法是让所有Car类型的实例都共享这个output()方法，这样如果有大批量的实例的话，就会节约很多内存。

解决这个问题，我们可以使用如下方式：

```
function Car(model, year, miles) {
    this.model = model;
    this.year = year;
    this.miles = miles;
    this.output= formatCar;
}
```



```
function formatCar() {
    return this.model + "走了" + this.miles + "公里";
}
```

这个方式虽然可用，但是我们有如下更好的方式。

构造函数与原型

JavaScript里函数有个原型属性叫prototype，当调用构造函数创建对象的时候，所有该构造函数原型的属性在新创建对象上都可用。按照这样，多个Car对象实例可以共享同一个原型，我们再扩展一下上例的代码：

```
function Car(model, year, miles) {
    this.model = model;
    this.year = year;
    this.miles = miles;
}

/*
注意：这里我们使用了Object.prototype.方法名，而不是Object.prototype
主要是用来避免重写定义原型prototype对象
*/
Car.prototype.output= function () {
    return this.model + "走了" + this.miles + "公里";
};

var tom = new Car("大叔", 2009, 20000);
var dudu = new Car("Dudu", 2010, 5000);

console.log(tom.output());
console.log(dudu.output());
```

这里，output()单实例可以在所有Car对象实例里共享使用。

另外：我们推荐构造函数以大写字母开头，以便区分普通的函数。

只能用new吗？

上面的例子对函数car都是用new来创建对象的，只有这一种方式么？其实还有别的方式，我们列举两种：

```
function Car(model, year, miles) {
    this.model = model;
    this.year = year;
    this.miles = miles;
    // 自定义一个output输出内容
    this.output = function () {
```

```

        return this.model + "走了" + this.miles + "公里";
    }
}

//方法1：作为函数调用
Car("大叔", 2009, 20000); //添加到window对象上
console.log(window.output());

//方法2：在另外一个对象的作用域内调用
var o = new Object();
Car.call(o, "Dudu", 2010, 5000);
console.log(o.output());

```

该代码的方法1有点特殊，如果不适用new直接调用函数的话，this指向的是全局对象window，我们来验证一下：

```

//作为函数调用
var tom = Car("大叔", 2009, 20000);
console.log(typeof tom); // "undefined"
console.log(window.output()); // "大叔走了20000公里"

```

这时候对象tom是undefined，而window.output()会正确输出结果，而如果使用new关键字则没有这个问题，验证如下：

```

//使用new 关键字
var tom = new Car("大叔", 2009, 20000);
console.log(typeof tom); // "object"
console.log(tom.output()); // "大叔走了20000公里"

```

强制使用new

上述的例子展示了不使用new的问题，那么我们有没有办法让构造函数强制使用new关键字呢，答案是肯定的，上代码：

```

function Car(model, year, miles) {
    if (!(this instanceof Car)) {
        return new Car(model, year, miles);
    }
    this.model = model;
    this.year = year;
    this.miles = miles;
    this.output = function () {
        return this.model + "走了" + this.miles + "公里";
    }
}

var tom = new Car("大叔", 2009, 20000);

```

```
var dudu = Car("Dudu", 2010, 5000);

console.log(typeof tom); // "object"
console.log(tom.output()); // "大叔走了20000公里"
console.log(typeof dudu); // "object"
console.log(dudu.output()); // "Dudu走了5000公里"
```

通过判断this的instanceof是不是Car来决定返回new Car还是继续执行代码，如果使用的是new关键字，则(this instanceof Car)为真，会继续执行下面的参数赋值，如果没有用new，(this instanceof Car)就为假，就会重新new一个实例返回。

原始包装函数

JavaScript里有3中原始包装函数：number, string, boolean，有时候两种都用：

```
// 使用原始包装函数
var s = new String("my string");
var n = new Number(101);
var b = new Boolean(true);

// 推荐这种
var s = "my string";
var n = 101;
var b = true;
```

推荐，只有在想保留数值状态的时候使用这些包装函数，关于区别可以参考下面的代码：

```
// 原始string
var greet = "Hello there";
// 使用split()方法分割
greet.split(' ')[0]; // "Hello"
// 给原始类型添加新属性不会报错
greet.smile = true;
// 单没法获取这个值（18章ECMAScript实现里我们讲了为什么）
console.log(typeof greet.smile); // "undefined"

// 原始string
var greet = new String("Hello there");
// 使用split()方法分割
greet.split(' ')[0]; // "Hello"
// 给包装函数类型添加新属性不会报错
greet.smile = true;
// 可以正常访问新属性
console.log(typeof greet.smile); // "boolean"
```

总结

本章主要讲解了构造函数模式的使用方法、调用方法以及new关键字的区别，希望大家在使

用的时候有所注意。

参

考：<http://www.addyosmani.com/resources/essentialjsdesignpatterns/book/#constructorpatternjavascript>

(27) 设计模式之建造者模式

介绍

在软件系统中，有时候面临着“一个复杂对象”的创建工作，其通常由各个部分的子对象用一定的算法构成；由于需求的变化，这个复杂对象的各个部分经常面临着剧烈的变化，但是将它们组合在一起的算法确相对稳定。如何应对这种变化？如何提供一种“封装机制”来隔离出“复杂对象的各个部分”的变化，从而保持系统中的“稳定构建算法”不随着需求改变而改变？这就是要说的建造者模式。

建造者模式可以将一个复杂对象的构建与其表示相分离，使得同样的构建过程可以创建不同的表示。也就是说如果我们用了建造者模式，那么用户就需要指定需要建造的类型就可以得到它们，而具体建造的过程和细节就不需要知道了。

正文

这个模式相对来说比较简单，先上代码，然后再解释

```
function getBeerById(id, callback) {
    // 使用ID来请求数据，然后返回数据。
    asyncRequest('GET', 'beer.uri?id=' + id, function (resp) {
        // callback调用 response
        callback(resp.responseText);
    });
}

var el = document.querySelector('#test');
el.addEventListener('click', getBeerByIdBridge, false);

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

根据建造者的定义，表相即是回调，也就是说获取数据以后如何显示和处理取决于回调函数，相应地回调函数在处理数据的时候不需要关注是如何获取数据的，同样的例子也可以在jquery的ajax方法里看到，有很多回调函数（比如success, error回调等），主要目的就是职责分离。

同样再来一个jQuery的例子：

```
$('<div class= "foo"> bar </div>');
```

我们只需要传入要生成的HTML字符，而不需要关系具体的HTML对象是如何生产的。

总结

建造者模式主要用于“分步骤构建一个复杂的对象”，在这其中“分步骤”是一个稳定的算法，而复杂对象的各个部分则经常变化，其优点是：建造者模式的“加工工艺”是暴露的，这样使得建造者模式更加灵活，并且建造者模式解耦了组装过程和创建具体部件，使得我们不用去关心每个部件是如何组装的。

参考：<http://www.cnblogs.com/feipeng/archive/2007/03/12/671932.html>

(28) 设计模式之工厂模式

介绍

与创建型模式类似，工厂模式创建对象（视为工厂里的产品）时无需指定创建对象的具体类。

工厂模式定义一个用于创建对象的接口，这个接口由子类决定实例化哪一个类。该模式使一个类的实例化延迟到了子类。而子类可以重写接口方法以便创建的时候指定自己的对象类型。

这个模式十分有用，尤其是创建对象的流程赋值的时候，比如依赖于很多设置文件等。并且，你会经常在程序里看到工厂方法，用于让子类类定义需要创建的对象类型。

正文

下面这个例子中，是应用了工厂方法对第26章构造函数模式代码的改进版本：

```
var Car = (function () {
    var Car = function (model, year, miles) {
        this.model = model;
        this.year = year;
        this.miles = miles;
    };
    return function (model, year, miles) {
        return new Car(model, year, miles);
    };
})();

var tom = new Car("Tom", 2009, 20000);
var dudu = new Car("Dudu", 2010, 5000);
```

不好理解的话，我们再给一个例子：

```
var productManager = {};

productManager.createProductA = function () {
    console.log('ProductA');
}

productManager.createProductB = function () {
    console.log('ProductB');
}

productManager.factory = function (typeType) {
```

```

        return new productManager[typeType];
    }

    productManager.factory("createProductA");

```

如果还不理解的话，那我们就再详细一点咯，假如我们想在网页面里插入一些元素，而这些元素类型不固定，可能是图片，也有可能是连接，甚至可能是文本，根据工厂模式的定义，我们需要定义工厂类和相应的子类，我们先来定义子类的具体实现（也就是子函数）：

```

var page = page || {};
page.dom = page.dom || {};
//子函数1：处理文本
page.dom.Text = function () {
    this.insert = function (where) {
        var txt = document.createTextNode(this.url);
        where.appendChild(txt);
    };
};

//子函数2：处理链接
page.dom.Link = function () {
    this.insert = function (where) {
        var link = document.createElement('a');
        link.href = this.url;
        link.appendChild(document.createTextNode(this.url));
        where.appendChild(link);
    };
};

//子函数3：处理图片
page.dom.Image = function () {
    this.insert = function (where) {
        var im = document.createElement('img');
        im.src = this.url;
        where.appendChild(im);
    };
};

```

那么我们如何定义工厂处理函数呢？其实很简单：

```

page.dom.factory = function (type) {
    return new page.dom[type];
}

```

使用方式如下：

```

var o = page.dom.factory('Link');

```



```
o.url = 'http://www.cnblogs.com';  
o.insert(document.body);
```

至此，工厂模式的介绍相信大家都已经了然于心了，我就不再多叙述了。

总结

什么时候使用工厂模式

以下几种情景下工厂模式特别有用：

1. 对象的构建十分复杂
2. 需要依赖具体环境创建不同实例
3. 处理大量具有相同属性的小对象

什么时候不该用工厂模式

不滥用运用工厂模式，有时候仅仅只是给代码增加了不必要的复杂度，同时使得测试难以运行下去。

(29) 设计模式之装饰者模式

介绍

装饰者提供比继承更有弹性的替代方案。装饰者用于包装同接口的对象，不仅允许你向方法添加行为，而且还可以将方法设置成原始对象调用（例如装饰者的构造函数）。

装饰者用于通过重载方法的形式添加新功能，该模式可以在被装饰者前面或者后面加上自己的行为以达到特定的目的。

正文

那么装饰者模式有什么好处呢？前面说了，装饰者是一种实现继承的替代方案。当脚本运行时，在子类中增加行为会影响原有类所有的实例，而装饰者却不然。取而代之的是它能给不同对象各自添加新行为。如下代码所示：

```
//需要装饰的类（函数）
function Macbook() {
    this.cost = function () {
        return 1000;
    };
}

function Memory(macbook) {
    this.cost = function () {
        return macbook.cost() + 75;
    };
}

function BlurayDrive(macbook) {
    this.cost = function () {
        return macbook.cost() + 300;
    };
}

function Insurance(macbook) {
    this.cost = function () {
        return macbook.cost() + 250;
    };
}

// 用法
var myMacbook = new Insurance(new BlurayDrive(new Memory(new Macbook())))
;
console.log(myMacbook.cost());
```

下面是另一个实例，当我们在装饰者对象上调用performTask时，它不仅具有一些装饰者的

行为，同时也调用了下层对象的performTask函数。

```
function ConcreteClass() {
    this.performTask = function () {
        this.preTask();
        console.log('doing something');
        this.postTask();
    };
}

function AbstractDecorator(decorated) {
    this.performTask = function () {
        decorated.performTask();
    };
}

function ConcreteDecoratorClass(decorated) {
    this.base = AbstractDecorator;
    this.base(decorated);

    decorated.preTask = function () {
        console.log('pre-calling..');
    };

    decorated.postTask = function () {
        console.log('post-calling..');
    };
}

var concrete = new ConcreteClass();
var decorator1 = new ConcreteDecoratorClass(concrete);
var decorator2 = new ConcreteDecoratorClass(decorator1);
decorator2.performTask();
```

再来一个彻底的例子：

```
var tree = {};
tree.decorate = function () {
    console.log('Make sure the tree won\'t fall');
};

tree.getDecorator = function (deco) {
    tree[deco].prototype = this;
    return new tree[deco];
};

tree.RedBalls = function () {
    this.decorate = function () {
        this.RedBalls.prototype.decorate(); // 第7步：先执行原型（这时候是Angel了）的decorate方法
        console.log('Put on some red balls'); // 第8步 再输出 red
```

```

        // 将这2步作为RedBalls的decorate方法
    }
};

tree.BlueBalls = function () {
    this.decorate = function () {
        this.BlueBalls.prototype.decorate(); // 第1步：先执行原型的decorate
        方法, 也就是tree.decorate()
        console.log('Add blue balls'); // 第2步 再输出blue
        // 将这2步作为BlueBalls的decorate方法
    }
};

tree.Angel = function () {
    this.decorate = function () {
        this.Angel.prototype.decorate(); // 第4步：先执行原型（这时候是BlueBa
        11s了）的decorate方法
        console.log('An angel on the top'); // 第5步 再输出angel
        // 将这2步作为Angel的decorate方法
    }
};

tree = tree.getDecorator('BlueBalls'); // 第3步：将BlueBalls对象赋给tree, 这
    时候父原型里的getDecorator依然可用
tree = tree.getDecorator('Angel'); // 第6步：将Angel对象赋给tree, 这时候父原型
    的父原型里的getDecorator依然可用
tree = tree.getDecorator('RedBalls'); // 第9步：将RedBalls对象赋给tree

tree.decorate(); // 第10步：执行RedBalls对象的decorate方法

```

总结

装饰者模式是为已有功能动态地添加更多功能的一种方式，把每个要装饰的功能放在单独的函数里，然后用该函数包装所要装饰的已有函数对象，因此，当需要执行特殊行为的时候，调用代码就可以根据需要有选择地、按顺序地使用装饰功能来包装对象。优点是把类（函数）的核心职责和装饰功能区分开了。

(30) 设计模式之外观模式

介绍

外观模式 (Facade) 为子系统的一组接口提供了一个一致的界面，此模块定义了一个高层接口，这个接口值得这一子系统更加容易使用。

正文

外观模式不仅简化类中的接口，而且对接口与调用者也进行了解耦。外观模式经常被认为开发者必备，它可以将一些复杂操作封装起来，并创建一个简单的接口用于调用。

外观模式经常被用于JavaScript类库里，通过它封装一些接口用于兼容多浏览器，外观模式可以让我们间接调用子系统，从而避免因直接访问子系统而产生不必要的错误。

外观模式的优势是易于使用，而且本身也比较轻量级。但也有缺点 外观模式被开发者连续使用时会产生一定的性能问题，因为在每次调用时都要检测功能的可用性。

下面是一段未优化过的代码，我们使用了外观模式通过检测浏览器特性的方式来创建一个跨浏览器的使用方法。

```
var addMyEvent = function (el, ev, fn) {  
    if (el.addEventListener) {  
        el.addEventListener(ev, fn, false);  
    } else if (el.attachEvent) {  
        el.attachEvent('on' + ev, fn);  
    } else {  
        el['on' + ev] = fn;  
    }  
};
```

再来一个简单的例子，说白了就是用接口封装其它的接口：

```
var mobileEvent = {  
    // ...  
    stop: function (e) {  
        e.preventDefault();  
        e.stopPropagation();  
    }  
    // ...  
};
```

总结

本文档使用 [看云](#) 构建

那么何时使用外观模式呢？一般来说分三个阶段：

首先，在设计初期，应该要有意识地将不同的两个层分离，比如经典的三层结构，在数据访问层和业务逻辑层、业务逻辑层和表示层之间建立外观Facade。

其次，在开发阶段，子系统往往因为不断的重构演化而变得越来越复杂，增加外观Facade可以提供一个简单的接口，减少他们之间的依赖。

第三，在维护一个遗留的大型系统时，可能这个系统已经很难维护了，这时候使用外观Facade也是非常合适的，为系系统开发一个外观Facade类，为设计粗糙和高度复杂的遗留代码提供比较清晰的接口，让新系统和Facade对象交互，Facade与遗留代码交互所有的复杂工作。

参考：大话设计模式

(31) 设计模式之代理模式

介绍

代理，顾名思义就是帮助别人做事，GoF对代理模式的定义如下：

代理模式（Proxy），为其他对象提供一种代理以控制对这个对象的访问。

代理模式使得代理对象控制具体对象的引用。代理几乎可以是任何对象：文件，资源，内存中的对象，或者是一些难以复制的东西。

正文

我们来举一个简单的例子，假如dudu要送酸奶小妹玫瑰花，却不知道她的联系方式或者不好意思，想委托大叔去送这些玫瑰，那大叔就是个代理（其实挺好的，可以扣几朵给媳妇），那我们如何做呢？

```
// 先声明美女对象
var girl = function (name) {
    this.name = name;
};

// 这是dudu
var dudu = function (girl) {
    this.girl = girl;
    this.sendGift = function (gift) {
        alert("Hi " + girl.name + ", dudu送你一个礼物：" + gift);
    }
};

// 大叔是代理
var proxyTom = function (girl) {
    this.girl = girl;
    this.sendGift = function (gift) {
        (new dudu(girl)).sendGift(gift); // 替dudu送花咯
    }
};
```

调用方式就非常简单了：

```
var proxy = new proxyTom(new girl("酸奶小妹"));
proxy.sendGift("999朵玫瑰");
```

实战一把

通过上面的代码，相信大家对代理模式已经非常清楚了，我们来实战下：我们有一个简单的播放列表，需要在点击单个连接（或者全选）的时候在该连接下方显示视频曲介绍以及play按钮，点击play按钮的时候播放视频，列表结构如下：

```
<h1>Dave Matthews vids</h1>
<p><span id="toggle-all">全选/反选</span></p>
<ol id="vids">
  <li><input type="checkbox" checked><a href="http://new.music.yahoo.com/videos/-2158073">Gravedigger</a></li>
  <li><input type="checkbox" checked><a href="http://new.music.yahoo.com/videos/-4472739">Save Me</a></li>
  <li><input type="checkbox" checked><a href="http://new.music.yahoo.com/videos/-45286339">Crush</a></li>
  <li><input type="checkbox" checked><a href="http://new.music.yahoo.com/videos/-2144530">Don't Drink The Water</a></li>
  <li><input type="checkbox" checked><a href="http://new.music.yahoo.com/videos/-217241800">Funny the Way It Is</a></li>
  <li><input type="checkbox" checked><a href="http://new.music.yahoo.com/videos/-2144532">What Would You Say</a></li>
</ol>
```

我们先来分析如下，首先我们不仅要监控a连接的点击事件，还要监控“全选/反选”的点击事件，然后请求服务器查询视频信息，组装HTML信息显示在li元素的最后位置上，效果如下：

1. ☒ [Gravedigger](#)



然后再监控play连接的点击事件，点击以后开始播放，效果如下：

1. ☒ [Gravedigger](#)



好了，开始，没有jQuery，我们自定义一个选择器：

```
var $ = function (id) {  
    return document.getElementById(id);  
};
```

由于Yahoo的json服务提供了callback参数，所以我们传入我们自定义的callback以便来接受数据，具体查询字符串拼装代码如下：

```
var http = {  
    makeRequest: function (ids, callback) {  
        var url = 'http://query.yahooapis.com/v1/public/yql?q=',  
            sql = 'select * from music.video.id where ids IN ("%ID%")',  
            format = "format=json",  
            handler = "***callback**=" + callback,  
            script = document.createElement('script');  
  
        sql = sql.replace('%ID%', ids.join('"', ''));  
        sql = encodeURIComponent(sql);  
  
        url += sql + '&' + format + '&' + handler;  
        script.src = url;
```

```

        document.body.appendChild(script);
    }
};

```

代理对象如下：

```

var proxy = {
    ids: [],
    delay: 50,
    timeout: null,
    callback: null,
    context: null,
    // 设置请求的id和callback以便在播放的时候触发回调
    makeRequest: function (id, callback, context) {

        // 添加到队列dd to the queue
        this.ids.push(id);

        this.callback = callback;
        this.context = context;

        // 设置timeout
        if (!this.timeout) {
            this.timeout = setTimeout(function () {
                proxy.flush();
            }, this.delay);
        }
    },
    // 触发请求，使用代理职责调用了http.makeRequest
    flush: function () {
        // proxy.handler为请求yahoo时的callback
        http.makeRequest(this.ids, 'proxy.handler');
        // 请求数据以后，紧接着执行proxy.handler方法（里面有另一个设置的callback）

        // 清楚timeout和队列
        this.timeout = null;
        this.ids = [];
    },
    handler: function (data) {
        var i, max;

        // 单个视频的callback调用
        if (parseInt(data.query.count, 10) === 1) {
            proxy.callback.call(proxy.context, data.query.results.Video);
            return;
        }

        // 多个视频的callback调用
        for (i = 0, max = data.query.results.Video.length; i < max; i +=
1) {
            proxy.callback.call(proxy.context, data.query.results.Video[i
]);
        }
    }
};

```

```

    }
  }
};

```

视频处理模块主要有3种子功能：获取信息、展示信息、播放视频：

```

var videos = {
  // 初始化播放器代码，开始播放
  getPlayer: function (id) {
    return '' +
      '<object width="400" height="255" id="uvp_fop" allowFullScree' +
      'n="true">' +
      '<param name="movie" value="http://d.yimg.com/m/up/fop/embedf' +
      'lv/swf/fop.swf"\/>' +
      '<param name="flashVars" value="id=v' + id + '&eID=130179' +
      '7&lang=us&enableFullScreen=0&shareEnable=1"\/>' +
      '<param name="wmode" value="transparent"\/>' +
      '<embed ' +
      'height="255" ' +
      'width="400" ' +
      'id="uvp_fop" ' +
      'allowFullScreen="true" ' +
      'src="http://d.yimg.com/m/up/fop/embedflv/swf/fop.swf" ' +
      'type="application/x-shockwave-flash" ' +
      'flashvars="id=v' + id + '&eID=1301797&lang=us&ym' +
      'psc=4195329&enableFullScreen=1&shareEnable=1"' +
      '\/>' +
      '<\object>';
  },
  // 拼接信息显示内容，然后在append到li的底部里显示
  updateList: function (data) {
    var id,
        html = '',
        info;

    if (data.query) {
      data = data.query.results.Video;
    }
    id = data.id;
    html += '';
    html += '<h2>' + data.title + '<\h2>';
    html += '<p>' + data.copyrightYear + ', ' + data.label + '<\p>';
    if (data.Album) {
      html += '<p>Album: ' + data.Album.Release.title + ', ' + data
        .Album.Release.releaseYear + '<br \/>';
    }
    html += '<p><a class="play" href="http://new.music.yahoo.com/vide' +
      'os/--' + id + '">&raquo; play<\a><\p>';
    info = document.createElement('div');
    info.id = "info" + id;
    info.innerHTML = html;
    $('v' + id).appendChild(info);
  },

```

```

// 获取信息并显示
getInfo: function (id) {
    var info = $('info' + id);

    if (!info) {
        proxy.makeRequest(id, videos.updateList, videos); //执行代理职
        责, 并传入videos.updateList回调函数
        return;
    }

    if (info.style.display === "none") {
        info.style.display = '';
    } else {
        info.style.display = 'none';
    }
}
};

```

现在可以处理点击事件的代码了，由于有很多a连接，如果每个连接都绑定事件的话，显然性能会有问题，所以我们将事件绑定在元素上，然后检测点击的是否是a连接，如果是说明我们点击的是视频地址，然后就可以播放了：

```

$('vids').onclick = function (e) {
    var src, id;

    e = e || window.event;
    src = e.target || e.srcElement;

    // 不是连接的话就不继续处理了
    if (src.nodeName.toUpperCase() !== "A") {
        return;
    }
    //停止冒泡
    if (typeof e.preventDefault === "function") {
        e.preventDefault();
    }
    e.returnValue = false;

    id = src.href.split('--')[1];

    //如果点击的是已经生产的视频信息区域的连接play, 就开始播放
    // 然后return不继续了
    if (src.className === "play") {
        src.parentNode.innerHTML = videos.getPlayer(id);
        return;
    }

    src.parentNode.id = "v" + id;
    videos.getInfo(id); // 这个才是第一次点击的时候显示视频信息的处理代码
};

```

全选反选的代码大同小异，我们就不解释了：

```
$('toggle-all').onclick = function (e) {
    var hrefs, i, max, id;

    hrefs = $('vids').getElementsByTagName('a');
    for (i = 0, max = hrefs.length; i < max; i += 1) {
        // 忽略play连接
        if (hrefs[i].className === "play") {
            continue;
        }
        // 忽略没有选择的项
        if (!hrefs[i].parentNode.firstChild.checked) {
            continue;
        }

        id = hrefs[i].href.split('--')[1];
        hrefs[i].parentNode.id = "v" + id;
        videos.getInfo(id);
    }
};
```

完整代码：<https://github.com/shichuan/javascript-patterns/blob/master/design-patterns/proxy.html>

总结

代理模式一般适用于如下场合：

1. 远程代理，也就是为了一个对象在不同的地址空间提供局部代表，这样可以隐藏一个对象存在于不同地址空间的事实，就像web service里的代理类一样。
2. 虚拟代理，根据需要创建开销很大的对象，通过它来存放实例化需要很长时间的真实对象，比如浏览器的渲染的时候先显示问题，而图片可以慢慢显示（就是通过虚拟代理代替了真实的图片，此时虚拟代理保存了真实图片的路径和尺寸。
3. 安全代理，用来控制真实对象访问时的权限，一般用于对象应该有不同访问权限。
4. 智能指引，只当调用真实的对象时，代理处理另外一些事情。例如C#里的垃圾回收，使用对象的时候会有引用次数，如果对象没有引用了，GC就可以回收它了。

参考：《大话设计模式》

(32) 设计模式之观察者模式

介绍

观察者模式又叫发布订阅模式 (Publish/Subscribe)，它定义了一种一对多的关系，让多个观察者对象同时监听某一个主题对象，这个主题对象的状态发生变化时就会通知所有的观察者对象，使得它们能够自动更新自己。

使用观察者模式的好处：

1. 支持简单的广播通信，自动通知所有已经订阅过的对象。
2. 页面载入后目标对象很容易与观察者存在一种动态关联，增加了灵活性。
3. 目标对象与观察者之间的抽象耦合关系能够单独扩展以及重用。

正文 (版本一)

JS里对观察者模式的实现是通过回调来实现的，我们来先定义一个pubsub对象，其内部包含了3个方法：订阅、退订、发布。

```
var pubsub = {};  
(function (q) {  
  
    var topics = {}, // 回调函数存放的数组  
        subUid = -1;  
    // 发布方法  
    q.publish = function (topic, args) {  
  
        if (!topics[topic]) {  
            return false;  
        }  
  
        setTimeout(function () {  
            var subscribers = topics[topic],  
                len = subscribers ? subscribers.length : 0;  
  
            while (len--) {  
                subscribers[len].func(topic, args);  
            }  
        }, 0);  
  
        return true;  
    };  
    //订阅方法  
    q.subscribe = function (topic, func) {  
  
        if (!topics[topic]) {
```

```

        topics[topic] = [];
    }

    var token = (++subUid).toString();
    topics[topic].push({
        token: token,
        func: func
    });
    return token;
};
//退订方法
q.unsubscribe = function (token) {
    for (var m in topics) {
        if (topics[m]) {
            for (var i = 0, j = topics[m].length; i < j; i++) {
                if (topics[m][i].token === token) {
                    topics[m].splice(i, 1);
                    return token;
                }
            }
        }
    }
    return false;
};
} (pubsub));

```

使用方式如下：

```

//来, 订阅一个
pubsub.subscribe('example1', function (topics, data) {
    console.log(topics + ": " + data);
});

//发布通知
pubsub.publish('example1', 'hello world!');
pubsub.publish('example1', ['test', 'a', 'b', 'c']);
pubsub.publish('example1', [{ 'color': 'blue' }, { 'text': 'hello' }]);

```

怎么样？用起来是不是很爽？但是这种方式有个问题，就是没办法退订订阅，要退订的话必须指定退订的名称，所以我们再来一个版本：

```

//将订阅赋值给一个变量，以便退订
var testSubscription = pubsub.subscribe('example1', function (topics, data) {
    console.log(topics + ": " + data);
});

//发布通知
pubsub.publish('example1', 'hello world!');
pubsub.publish('example1', ['test', 'a', 'b', 'c']);

```



```

pubsub.publish('example1', [{ 'color': 'blue' }, { 'text': 'hello' }]);

//退订
setTimeout(function () {
    pubsub.unsubscribe(testSubscription);
}, 0);

//再发布一次, 验证一下是否还能够输出信息
pubsub.publish('example1', 'hello again! (this will fail)');

```

版本二

我们也可以利用原型的特性实现一个观察者模式, 代码如下:

```

function Observer() {
    this.fns = [];
}
Observer.prototype = {
    subscribe: function (fn) {
        this.fns.push(fn);
    },
    unsubscribe: function (fn) {
        this.fns = this.fns.filter(
            function (el) {
                if (el !== fn) {
                    return el;
                }
            }
        );
    },
    update: function (o, thisObj) {
        var scope = thisObj || window;
        this.fns.forEach(
            function (el) {
                el.call(scope, o);
            }
        );
    }
};

//测试
var o = new Observer;
var f1 = function (data) {
    console.log('Robbin: ' + data + ', 赶紧干活了!');
};

var f2 = function (data) {
    console.log('Randall: ' + data + ', 找他加点工资去!');
};

o.subscribe(f1);
o.subscribe(f2);

o.update("Tom回来了!")

```

```
//退订f1
o.unsubscribe(f1);
//再来验证
o.update("Tom回来了!");
```

如果提示找不到filter或者forEach函数，可能是因为你的浏览器还不够新，暂时不支持新标准的函数，你可以使用如下方式自己定义：

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

版本三

如果想让多个对象都具有观察者发布订阅的功能，我们可以定义一个通用的函数，然后将该函数的功能应用到需要观察者功能的对象上，代码如下：

```
//通用代码
var observer = {
    //订阅
    addSubscriber: function (callback) {
        this.subscribers[this.subscribers.length] = callback;
    },
    //退订
    removeSubscriber: function (callback) {
        for (var i = 0; i < this.subscribers.length; i++) {
            if (this.subscribers[i] === callback) {
                delete (this.subscribers[i]);
            }
        }
    },
}
```

```

//发布
publish: function (what) {
    for (var i = 0; i < this.subscribers.length; i++) {
        if (typeof this.subscribers[i] === 'function') {
            this.subscribers[i](what);
        }
    }
},
// 将对象o具有观察者功能
make: function (o) {
    for (var i in this) {
        o[i] = this[i];
        o.subscribers = [];
    }
}
};

```

然后订阅2个对象blogger和user，使用observer.make方法将这2个对象具有观察者功能，代码如下：

```

var blogger = {
    recommend: function (id) {
        var msg = 'dudu 推荐了的帖子:' + id;
        this.publish(msg);
    }
};

var user = {
    vote: function (id) {
        var msg = '有人投票了!ID=' + id;
        this.publish(msg);
    }
};

observer.make(blogger);
observer.make(user);

```

使用方法就比较简单了，订阅不同的回调函数，以便可以注册到不同的观察者对象里（也可以同时注册到多个观察者对象里）：

```

var tom = {
    read: function (what) {
        console.log('Tom看到了如下信息:' + what)
    }
};

var mm = {
    show: function (what) {
        console.log('mm看到了如下信息:' + what)
    }
}

```

```

};
// 订阅
blogger.addSubscriber(tom.read);
blogger.addSubscriber(mm.show);
blogger.recommend(123); //调用发布

//退订
blogger.removeSubscriber(mm.show);
blogger.recommend(456); //调用发布

//另外一个对象的订阅
user.addSubscriber(mm.show);
user.vote(789); //调用发布

```

jQuery版本

根据jQuery1.7版新增的on/off功能，我们也可以定义jQuery版的观察者：

```

(function ($) {

    var o = $({});

    $.subscribe = function () {
        o.on.apply(o, arguments);
    };

    $.unsubscribe = function () {
        o.off.apply(o, arguments);
    };

    $.publish = function () {
        o.trigger.apply(o, arguments);
    };

} (jQuery));

```

调用方法比上面3个版本都简单：

```

//回调函数
function handle(e, a, b, c) {
    // `e`是事件对象，不需要关注
    console.log(a + b + c);
};

//订阅
$.subscribe("/some/topic", handle);
//发布
$.publish("/some/topic", ["a", "b", "c"]); // 输出abc

$.unsubscribe("/some/topic", handle); // 退订

```

```
//订阅
$.subscribe("/some/topic", function (e, a, b, c) {
    console.log(a + b + c);
});

$.publish("/some/topic", ["a", "b", "c"]); // 输出abc

//退订（退订使用的是/some/topic名称，而不是回调函数哦，和版本一的例子不一样）
$.unsubscribe("/some/topic");
```

可以看到，他的订阅和退订使用的是字符串名称，而不是回调函数名称，所以即便传入的是匿名函数，我们也是可以退订的。

总结

观察者的使用场合就是：当一个对象的改变需要同时改变其它对象，并且它不知道具体有多少对象需要改变的时候，就应该考虑使用观察者模式。

总的来说，观察者模式所做的工作就是在解耦，让耦合的双方都依赖于抽象，而不是依赖于具体。从而使得各自的变化都不会影响到另一边的变化。

参考地址：

<https://github.com/shichuan/javascript-patterns/blob/master/design-patterns/observer.html>

<http://www.addyosmani.com/resources/essentialjsdesignpatterns/book/#observerpatternjavascript>

<https://gist.github.com/661855>

(33) 设计模式之策略模式

介绍

策略模式定义了算法家族，分别封装起来，让他们之间可以互相替换，此模式让算法的变化不会影响到使用算法的客户。

正文

在理解策略模式之前，我们先来一个例子，一般情况下，如果我们要做数据合法性验证，很多时候都是按照switch语句来判断，但是这就带来几个问题，首先如果增加需求的话，我们还要再次修改这段代码以增加逻辑，而且在进行单元测试的时候也会越来越复杂，代码如下：

```
validator = {
  validate: function (value, type) {
    switch (type) {
      case 'isNonEmpty ':
        {
          return true; // NonEmpty 验证结果
        }
      case 'isNumber ':
        {
          return true; // Number 验证结果
          break;
        }
      case 'isAlphaNum ':
        {
          return true; // AlphaNum 验证结果
        }
      default:
        {
          return true;
        }
    }
  }
};
// 测试
alert(validator.validate("123", "isNonEmpty"));
```

那如何来避免上述代码中的问题呢，根据策略模式，我们可以将相同的工作代码单独封装成不同的类，然后通过统一的策略处理类来处理，OK，我们先来定义策略处理类，代码如下：

```
var validator = {

  // 所有可以的验证规则处理类存放的地方，后面会单独定义
  types: {},
```

```

// 验证类型所对应的错误消息
messages: [],

// 当然需要使用的验证类型
config: {},

// 暴露的公开验证方法
// 传入的参数是 key => value对
validate: function (data) {

    var i, msg, type, checker, result_ok;

    // 清空所有的错误信息
    this.messages = [];

    for (i in data) {
        if (data.hasOwnProperty(i)) {

            type = this.config[i]; // 根据key查询是否有存在的验证规则
            checker = this.types[type]; // 获取验证规则的验证类

            if (!type) {
                continue; // 如果验证规则不存在, 则不处理
            }
            if (!checker) { // 如果验证规则类不存在, 抛出异常
                throw {
                    name: "ValidationError",
                    message: "No handler to validate type " + type
                };
            }

            result_ok = checker.validate(data[i]); // 使用查到的单个验证类进行验证
            if (!result_ok) {
                msg = "Invalid value for *" + i + "*", " + checker.instructions;
                this.messages.push(msg);
            }
        }
    }
    return this.hasErrors();
},

// helper
hasErrors: function () {
    return this.messages.length !== 0;
}
};

```

然后剩下的工作, 就是定义types里存放的各种验证类了, 我们这里只举几个例子:

```
// 验证给定的值是否不为空
```

```

validator.types.isEmpty = {
  validate: function (value) {
    return value !== "";
  },
  instructions: "传入的值不能为空"
};

// 验证给定的值是否是数字
validator.types.isNumber = {
  validate: function (value) {
    return !isNaN(value);
  },
  instructions: "传入的值只能是合法的数字，例如：1, 3.14 or 2010"
};

// 验证给定的值是否只是字母或数字
validator.types.isAlphaNum = {
  validate: function (value) {
    return !/^[a-z0-9]/i.test(value);
  },
  instructions: "传入的值只能保护字母和数字，不能包含特殊字符"
};

```

使用的时候，我们首先要定义需要验证的数据集合，然后还需要定义每种数据需要验证的规则类型，代码如下：

```

var data = {
  first_name: "Tom",
  last_name: "Xu",
  age: "unknown",
  username: "TomXu"
};

validator.config = {
  first_name: 'isEmpty',
  age: 'isNumber',
  username: 'isAlphaNum'
};

```

最后，获取验证结果的代码就简单了：

```

validator.validate(data);

if (validator.hasErrors()) {
  console.log(validator.messages.join("\n"));
}

```

总结

策略模式定义了一系列算法，从概念上来说，所有的这些算法都是做相同的事情，只是实现不同，他可以以相同的方式调用所有的方法，减少了各种算法类与使用算法类之间的耦合。

从另外一个层面上来说，单独定义算法类，也方便了单元测试，因为可以通过自己的算法进行单独测试。

实践中，不仅可以封装算法，也可以用来封装几乎任何类型的规则，是要在分析过程中需要在不同时间应用不同的业务规则，就可以考虑是要策略模式来处理各种变化。

(34) 设计模式之命令模式

介绍

命令模式(Command)的定义是：用于将一个请求封装成一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或者记录请求日志，以及执行可撤销的操作。也就是说改模式旨在将函数的调用、请求和操作封装成一个单一的对象，然后对这个对象进行一系列的处理。此外，可以通过调用实现具体函数的对象来解耦命令对象与接收对象。

正文

我们来通过车辆购买程序来展示这个模式，首先定义车辆购买的具体操作类：

```
$(function () {  
    var CarManager = {  
        // 请求信息  
        requestInfo: function (model, id) {  
            return 'The information for ' + model +  
                ' with ID ' + id + ' is foobar';  
        },  
        // 购买汽车  
        buyVehicle: function (model, id) {  
            return 'You have successfully purchased Item '  
                + id + ', a ' + model;  
        },  
        // 组织view  
        arrangeViewing: function (model, id) {  
            return 'You have successfully booked a viewing of '  
                + model + ' ( ' + id + ' ) '  
            }  
        };  
    })();
```

来看一下上述代码，通过调用函数来简单执行manager的命令，然而在一些情况下，我们并不想直接调用对象内部的方法。这样会增加对象与对象间的依赖。现在我们来扩展一下这个CarManager 使其能够接受任何来自包括model和car ID 的CarManager对象的处理请求。根据命令模式的定义，我们希望实现如下这种功能的调用：

```
CarManager.execute({ commandType: "buyVehicle", operand1: 'Ford Escort',  
    operand2: '453543' });
```

根据这样的需求，我们可以这样啦实现CarManager.execute方法：

```
CarManager.execute = function (command) {  
    return CarManager[command.request](command.model, command.carID);  
};
```

改造以后，调用就简单多了，如下调用都可以实现（当然有些异常细节还是需要再完善一下的）：

```
CarManager.execute({ request: "arrangeViewing", model: 'Ferrari', carID:  
'145523' });  
CarManager.execute({ request: "requestInfo", model: 'Ford Mondeo', carID:  
'543434' });  
CarManager.execute({ request: "requestInfo", model: 'Ford Escort', carID:  
'543434' });  
CarManager.execute({ request: "buyVehicle", model: 'Ford Escort', carID:  
'543434' });
```

总结

命令模式比较容易设计一个命令队列，在需求的情况下比较容易将命令计入日志，并且允许接受请求的一方决定是否调用，而且可以实现对请求的撤销和重设，而且由于新增的具体类不影响其他的类，所以很容易实现。

但敏捷开发原则告诉我们，不要为代码添加基于猜测的、实际不需要的功能，如果不清楚一个系统是否需要命令模式，一般就不要着急去实现它，事实上，在需求的时候通过重构实现这个模式并不困难，只有在真正需求如撤销、恢复操作等功能时，把原来的代码重构为命令模式才有意义。

(35) 设计模式之迭代器模式

介绍

迭代器模式(Iterator)：提供一种方法顺序一个聚合对象中各个元素，而又不暴露该对象内部表示。

迭代器的几个特点是：

1. 访问一个聚合对象的内容而无需暴露它的内部表示。
2. 为遍历不同的集合结构提供一个统一的接口，从而支持同样的算法在不同的集合结构上进行操作。
3. 遍历的同时更改迭代器所在的集合结构可能会导致问题（比如C#的foreach里不允许修改item）。

正文

一般的迭代，我们至少要有2个方法，hasNext()和Next()，这样才做做到遍历所有对象，我们先给出一个例子：

```
var agg = (function () {
    var index = 0,
        data = [1, 2, 3, 4, 5],
        length = data.length;

    return {
        next: function () {
            var element;
            if (!this.hasNext()) {
                return null;
            }
            element = data[index];
            index = index + 1;
            return element;
        },

        hasNext: function () {
            return index < length;
        },

        rewind: function () {
            index = 0;
        },

        current: function () {
            return data[index];
        }
    };
})();
```

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

使用方法和平时C#里的方式是一样的：

```
// 迭代的结果是：1, 3, 5  
while (agg.hasNext()) {  
    console.log(agg.next());  
}
```

当然，你也可以通过额外的方法来重置数据，然后再继续其它操作：

```
// 重置  
agg.rewind();  
console.log(agg.current()); // 1
```

jQuery应用例子

jQuery里一个非常有名的迭代器就是\$.each方法，通过each我们可以传入额外的function，然后来对所有的item项进行迭代操作，例如：

```
$.each(['dudu', 'dudu', '酸奶小妹', '那个MM'], function (index, value) {  
    console.log(index + ': ' + value);  
});  
//或者  
$('li').each(function (index) {  
    console.log(index + ': ' + $(this).text());  
});
```

总结

迭代器的使用场景是：对于集合内部结果常常变化各异，我们不想暴露其内部结构的话，但又响让客户代码透明底访问其中的元素，这种情况下我们可以使用迭代器模式。

(36) 设计模式之中介者模式

介绍

中介者模式 (Mediator) , 用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显式地相互引用, 从而使其耦合松散, 而且可以独立地改变它们之间的交互。

主要内容来

自 : <http://www.addyosmani.com/resources/essentialjsdesignpatterns/book/#mediatorpatternjavascript>

正文

软件开发中, 中介者是一个行为设计模式, 通过提供一个统一的接口让系统的不同部分进行通信。一般, 如果系统有很多子模块需要直接沟通, 都要创建一个中央控制点让其各模块通过该中央控制点进行交互。中介者模式可以让这些子模块不需要直接沟通, 而达到进行解耦的目的。

打个比方, 平时常见的机场交通控制系统, 塔台就是中介者, 它控制着飞机 (子模块) 的起飞和降落, 因为所有的沟通都是从飞机向塔台汇报来完成的, 而不是飞机之前相互沟通。中央控制系统就是该系统的关键, 也就是软件设计中扮演的中介者角色。

我们先用伪代码来理解一下 :

```
// 如下代码是伪代码, 请不要过分在意代码
// 这里app命名空间就相当于扮演中介者的角色
var app = app || {};

// 通过app中介者来进行Ajax请求
app.sendRequest = function ( options ) {
    return $.ajax($.extend({}, options));
}

// 请求URL以后, 展示View
app.populateView = function( url, view ){
    $.when(app.sendRequest({url: url, method: 'GET'}))
        .then(function(){
            //显示内容
        });
}

// 清空内容
app.resetView = function( view ){
    view.html('');
}
```

本文档使用 看云 构建

在JavaScript里，中介者非常常见，相当于观察者模式上的消息Bus，只不过不像观察者那样通过调用pub/sub的形式来实现，而是通过中介者统一来管理，让我们在观察者的基础上来给出一个例子：

```
var mediator = (function () {
    // 订阅一个事件，并且提供一个事件触发以后的回调函数
    var subscribe = function (channel, fn) {
        if (!mediator.channels[channel]) mediator.channels[channel] = [];
        mediator.channels[channel].push({ context: this, callback: fn });
        return this;
    },

    // 广播事件
    publish = function (channel) {
        if (!mediator.channels[channel]) return false;
        var args = Array.prototype.slice.call(arguments, 1);
        for (var i = 0, l = mediator.channels[channel].length; i < l; i++)
        ) {
            var subscription = mediator.channels[channel][i];
            subscription.callback.apply(subscription.context, args);
        }
        return this;
    };

    return {
        channels: {},
        publish: publish,
        subscribe: subscribe,
        installTo: function (obj) {
            obj.subscribe = subscribe;
            obj.publish = publish;
        }
    };
})();
```

调用代码，相对就简单了：

```
(function (Mediator) {

    function initialize() {

        // 默认值
        mediator.name = "dudu";

        // 订阅一个事件nameChange
        // 回调函数显示修改前后的信息
        mediator.subscribe('nameChange', function (arg) {
            console.log(this.name);
        });
    }
})();
```

```

        this.name = arg;
        console.log(this.name);
    });
}

function updateName() {
    // 广播触发事件, 参数为新数据
    mediator.publish('nameChange', 'tom'); // dudu, tom
}

initialize(); // 初始化
updateName(); // 调用

})(mediator);

```

中介者和观察者

到这里, 大家可能迷糊了, 中介者和观察者貌似差不多, 有什么不同呢? 其实是有点类似, 但是我们来看看具体的描述:

观察者模式, 没有封装约束的单个对象, 相反, 观察者Observer和具体类Subject是一起配合来维护约束的, 沟通是通过多个观察者和多个具体类来交互的: 每个具体类通常包含多个观察者, 而有时候具体类里的一个观察者也是另一个观察者的具体类。

而中介者模式所做的不是简单的分发, 却是扮演着维护这些约束的职责。

中介者和外观模式

很多人可能也比较迷糊中介者和外观模式的区别, 他们都是对现有各模块进行抽象, 但有一些微妙的区别。

中介者所做的是在模块之间进行通信, 是多向的, 但外观模式只是为某一个模块或系统定义简单的接口而不添加额外的功能。系统中的其它模块和外观模式这个概念没有直接联系, 可以认为是单向性。

完整的例子

再给出一个完整的例子:

```

<!doctype html>
<html lang="en">
<head>
    <title>JavaScript Patterns</title>
    <meta charset="utf-8">
</head>
<body>
<div id="results"></div>
    <script>
        function Player(name) {

```



```

        this.points = 0;
        this.name = name;
    }
    Player.prototype.play = function () {
        this.points += 1;
        mediator.played();
    };
    var scoreboard = {

        // 显示内容的容器
        element: document.getElementById('results'),

        // 更新分数显示
        update: function (score) {
            var i, msg = '';
            for (i in score) {
                if (score.hasOwnProperty(i)) {
                    msg += '<p><strong>' + i + '<\strong>: ';
                    msg += score[i];
                    msg += '<\p>';
                }
            }
            this.element.innerHTML = msg;
        }
    };

    var mediator = {

        // 所有的player
        players: {},

        // 初始化
        setup: function () {
            var players = this.players;
            players.home = new Player('Home');
            players.guest = new Player('Guest');
        },

        // play以后, 更新分数
        played: function () {
            var players = this.players,
                score = {
                    Home: players.home.points,
                    Guest: players.guest.points
                };

            scoreboard.update(score);
        },

        // 处理用户按键交互
        keypress: function (e) {
            e = e || window.event; // IE
            if (e.which === 49) { // 数字键 "1"
                mediator.players.home.play();
                return;
            }
        }
    };

```

```
                if (e.which === 48) { // 数字键 "0"
                    mediator.players.guest.play();
                    return;
                }
            };

            // go!
            mediator.setup();
            window.onkeypress = mediator.keypress;

            // 30秒以后结束
            setTimeout(function () {
                window.onkeypress = null;
                console.log('Game over!');
            }, 30000);
        </script>
    </body>
</html>
```

总结

中介者模式一般应用于一组对象已定义良好但是以复杂的方式进行通信的场合，一般情况下，中介者模式很容易在系统中使用，但也容易在系统里误用，当系统出现了多对多交互复杂的对象群时，先不要急于使用中介者模式，而是要思考一下是不是系统设计有问题。

另外，由于中介者模式把交互复杂性变成了中介者本身的复杂性，所以说中介者对象会比其它任何对象都复杂。

(37) 设计模式之享元模式

介绍

享元模式 (Flyweight) , 运行共享技术有效地支持大量细粒度的对象 , 避免大量拥有相同内容的小类的开销(如耗费内存) , 使大家共享一个类(元类)。

享元模式可以避免大量非常相似类的开销 , 在程序设计中 , 有时需要生产大量细粒度的类实例来表示数据 , 如果能发现这些实例除了几个参数以外 , 开销基本相同的话 , 就可以大幅度减少需要实例化的类的数量。如果能把那些参数移动到类实例的外面 , 在方法调用的时候将他们传递进来 , 就可以通过共享大幅度减少单个实例的数目。

那么如果在JavaScript中应用享元模式呢? 有两种方式 , 第一种是应用在数据层上 , 主要是应用在内存里大量相似的对象上 ; 第二种是应用在DOM层上 , 享元可以用在中央事件管理器上用来避免给父容器里的每个子元素都附加事件句柄。

享元与数据层

Flyweight中有两个重要概念--内部状态intrinsic和外部状态extrinsic之分 , 内部状态就是在对象里通过内部方法管理 , 而外部信息可以在通过外部删除或者保存。

说白了,就是先捏一个的原始模型 , 然后随着不同场合和环境,再产生各具特征的具体模型 , 很显然,在这里需要产生不同的新对象 , 所以Flyweight模式中常出现Factory模式 , Flyweight的内部状态是用来共享的 , Flyweight factory负责维护一个Flyweight pool(模式池)来存放内部状态的对象。

使用享元模式

让我们来演示一下如果通过一个类库让系统来管理所有的书籍 , 每个书籍的元数据暂定为如下内容 :

```
ID
Title
Author
Genre
Page count
Publisher ID
ISBN
```

我们还需要定义每本书被借出去的时间和借书人 , 以及退书日期和是否可用状态 :

```

checkoutDate
checkoutMember
dueReturnDate
availability

```

因为book对象设置成如下代码，注意该代码还未被优化：

```

var Book = function( id, title, author, genre, pageCount,publisherID, ISBN
N, checkoutDate, checkoutMember, dueReturnDate,availability ){
    this.id = id;
    this.title = title;
    this.author = author;
    this.genre = genre;
    this.pageCount = pageCount;
    this.publisherID = publisherID;
    this.ISBN = ISBN;
    this.checkoutDate = checkoutDate;
    this.checkoutMember = checkoutMember;
    this.dueReturnDate = dueReturnDate;
    this.availability = availability;
};
Book.prototype = {
    getTitle:function(){
        return this.title;
    },
    getAuthor: function(){
        return this.author;
    },
    getISBN: function(){
        return this.ISBN;
    },
    /*其它get方法在这里就不显示了*/

    // 更新借出状态
    updateCheckoutStatus: function(bookID, newStatus, checkoutDate,checkoutMe
mber, newReturnDate){
        this.id = bookID;
        this.availability = newStatus;
        this.checkoutDate = checkoutDate;
        this.checkoutMember = checkoutMember;
        this.dueReturnDate = newReturnDate;
    },
    //续借
    extendCheckoutPeriod: function(bookID, newReturnDate){
        this.id = bookID;
        this.dueReturnDate = newReturnDate;
    },
    //是否到期
    isPastDue: function(bookID){
        var currentDate = new Date();
        return currentDate.getTime() > Date.parse(this.dueReturnDate);
    }
};

```

程序刚开始可能没问题，但是随着时间的增加，图书可能大批量增加，并且每种图书都有不同的版本和数量，你将会发现系统变得越来越慢。几千个book对象在内存里可想而知，我们需要用享元模式来优化。

我们可以将数据分成内部和外部两种数据，和book对象相关的数据（title, author 等）可以归结为内部属性，而（checkoutMember, dueReturnDate等）可以归结为外部属性。这样，如下代码就可以在同一本书里共享同一个对象了，因为不管谁借的书，只要书是同一本书，基本信息是一样的：

```
/*享元模式优化代码*/
var Book = function(title, author, genre, pageCount, publisherID, ISBN){
    this.title = title;
    this.author = author;
    this.genre = genre;
    this.pageCount = pageCount;
    this.publisherID = publisherID;
    this.ISBN = ISBN;
};
```

定义基本工厂

让我们来定义一个基本工厂，用来检查之前是否创建该book的对象，如果有就返回，没有就重新创建并存储以便后面可以继续访问，这确保我们为每一种书只创建一个对象：

```
/* Book工厂 单例 */
var BookFactory = (function(){
    var existingBooks = {};
    return{
        createBook: function(title, author, genre,pageCount,publisherID,ISBN){
            /*查找之前是否创建*/
            var existingBook = existingBooks[ISBN];
            if(existingBook){
                return existingBook;
            }else{
                /* 如果没有，就创建一个，然后保存*/
                var book = new Book(title, author, genre,pageCount,publisherID,ISBN);
                existingBooks[ISBN] = book;
                return book;
            }
        }
    };
});
```

管理外部状态

本文档使用 [看云](#) 构建

外部状态，相对就简单了，除了我们封装好的book，其它都需要在这里管理：

```

/*BookRecordManager 借书管理类 单例*/
var BookRecordManager = (function(){
    var bookRecordDatabase = {};
    return{
        /*添加借书记录*/
        addBookRecord: function(id, title, author, genre, pageCount, publisherID, ISBN, checkoutDate, checkoutMember, dueReturnDate, availability){
            var book = bookFactory.createBook(title, author, genre, pageCount, publisherID, ISBN);
            bookRecordDatabase[id] = {
                checkoutMember: checkoutMember,
                checkoutDate: checkoutDate,
                dueReturnDate: dueReturnDate,
                availability: availability,
                book: book;
            };
        },
        updateCheckoutStatus: function(bookID, newStatus, checkoutDate, checkoutMember, newReturnDate){
            var record = bookRecordDatabase[bookID];
            record.availability = newStatus;
            record.checkoutDate = checkoutDate;
            record.checkoutMember = checkoutMember;
            record.dueReturnDate = newReturnDate;
        },
        extendCheckoutPeriod: function(bookID, newReturnDate){
            bookRecordDatabase[bookID].dueReturnDate = newReturnDate;
        },
        isPastDue: function(bookID){
            var currentDate = new Date();
            return currentDate.getTime() > Date.parse(bookRecordDatabase[bookID].dueReturnDate);
        }
    };
})();

```

通过这种方式，我们做到了将同一种图书的相同信息保存在一个bookmanager对象里，而且只保存一份；相比之前的代码，就可以发现节约了很多内存。

享元模式与DOM

关于DOM的事件冒泡，在这里就不多说了，相信大家已经知道了，我们举两个例子。

例1：事件集中管理

举例来说，如果我们有大量相似类型的元素或者结构（比如菜单，或者ul里的多个li）都需要监控他的click事件的话，那就需要多每个元素进行事件绑定，如果元素有非常非常多，那性能就可想而知了，而结合冒泡的知识，任何一个子元素有事件触发的话，那触发以后事件将

冒泡到上一级元素，所以利用这个特性，我们可以使用享元模式，我们可以对这些相似元素的父级元素进行事件监控，然后再判断里面哪个子元素有事件触发了，再进行进一步的操作。

在这里我们结合一下jQuery的bind/unbind方法来举例。

HTML:

```
<div id="container">
  <div class="toggle" href="#">更多信息（地址）
    <span class="info">
      这里是更多信息
    </span></div>
  <div class="toggle" href="#">更多信息（地图）
    <span class="info">
      <iframe src="http://www.map-generator.net/extmap.php?name=Londo
n&address=london%2C%20england&width=500...gt;"></iframe>
    </span>
  </div>
</div>
```

JavaScript:

```
stateManager = {
  fly: function(){
    var self = this;
    $('#container').unbind().bind("click", function(e){
      var target = $(e.originalTarget || e.srcElement);
      // 判断是哪一个子元素
      if(target.is("div.toggle")){
        self.handleClick(target);
      }
    });
  },

  handleClick: function(elem){
    elem.find('span').toggle('slow');
  }
};
```

例2：应用享元模式提升性能

另外一个例子，依然和jQuery有关，一般我们在事件的回调函数里使用元素对象是会后，经常会用到\$(this)这种形式，其实它重复创建了新对象，因为本身回调函数里的this已经是DOM元素自身了，我们必要必要使用如下这样的代码：

```
$('#div').bind('click', function(){
  本文档使用 看云 构建
```

```

    console.log('You clicked: ' + $(this).attr('id'));
  });
// 上面的代码, 要避免使用, 避免再次对DOM元素进行生成jQuery对象, 因为这里可以直接使用DOM元素自身了。
$('div').bind('click', function(){
    console.log('You clicked: ' + this.id);
});

```

其实, 如果非要用\$(this)这样的形式, 我们也可以实现自己版本的单实例模式, 比如我们来实现一个jQuery.single(this)这样的函数以便返回DOM元素自身:

```

jQuery.single = (function(o){

    var collection = jQuery([1]);
    return function(element) {

        // 将元素放到集合里
        collection[0] = element;

        // 返回集合
        return collection;

    };

});

```

使用方法:

```

$('div').bind('click', function(){
    var html = jQuery.single(this).next().html();
    console.log(html);
});

```

这样, 就是原样返回DOM元素自身了, 而且不进行jQuery对象的创建。

总结

Flyweight模式是一个提高程序效率和性能的模式, 会大大加快程序的运行速度. 应用场合很多: 比如你要从一个数据库中读取一系列字符串, 这些字符串中有许多是重复的, 那么我们可以将这些字符串储存在Flyweight池(pool)中。

如果一个应用程序使用了大量的对象, 而这些大量的对象造成了很大的存储开销时就应该考虑使用享元模式; 还有就是对象的大多数状态可以外部状态, 如果删除对象的外部状态, 那么就可以用相对较少的共享对象取代很多组对象, 此时可以考虑使用享元模式。

参考地

址：<http://www.addyosmani.com/resources/essentialjsdesignpatterns/book/#detailflyweight>

(38) 设计模式之职责链模式

介绍

职责链模式 (Chain of responsibility) 是使多个对象都有机会处理请求，从而避免请求的发送者和接受者之间的耦合关系。将这个对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理他为止。

也就是说，请求以后，从第一个对象开始，链中收到请求的对象要么亲自处理它，要么转发给链中的下一个候选者。提交请求的对象并不明确知道哪一个对象将会处理它——也就是该请求有一个隐式的接受者 (implicit receiver)。根据运行时刻，任一候选者都可以响应相应的请求，候选者的数目是任意的，你可以在运行时刻决定哪些候选者参与到链中。

正文

对于JavaScript实现，我们可以利用其原型特性来实现职责链模式。

```
var NO_TOPIC = -1;
var Topic;

function Handler(s, t) {
    this.successor = s || null;
    this.topic = t || 0;
}

Handler.prototype = {
    handle: function () {
        if (this.successor) {
            this.successor.handle()
        }
    },
    has: function () {
        return this.topic != NO_TOPIC;
    }
};
```

Handler只是接受2个参数，第一个是继任者（用于将处理请求传下去），第二个是传递层级（可以用于控制在某个层级下是否执行某个操作，也可以不用），Handler原型暴露了一个handle方法，这是实现该模式的重点，先来看看如何使用上述代码。

```
var app = new Handler({
    handle: function () {
        console.log('app handle');
    }
});
```

```

    }, 3);

    var dialog = new Handler(app, 1);

    var button = new Handler(dialog, 2);

    button.handle();

```

改代码通过原型特性，调用代码从button.handle()->dialog.handle()->app.handle()->参数里的handle()，前三个都是调用原型的handle，最后才查找到传入的参数里的handle，然后输出结果，也就是说其实只有最后一层才处理。

那如何做到调用的时候，只让dialog的这个对象进行处理呢？其实可以定义dialog实例对象的handle方法就可以了，但需要在new button的之前来做，代码如下：

```

var app = new Handler({
    handle: function () {
        console.log('app handle');
    }
}, 3);

var dialog = new Handler(app, 1);
dialog.handle = function () {
    console.log('dialog before ...')
    // 这里做具体的处理操作
    console.log('dialog after ...')
};

var button = new Handler(dialog, 2);

button.handle();

```

该代码的执行结果即时dialog.handle里的处理结果，而不再是给app传入的参数里定义的handle的执行操作。

那能不能做到自身处理完以后，然后在让继任者继续处理呢？答案是肯定的，但是在调用的handle以后，需要利用原型的特性调用如下代码：

```

Handler.prototype.handle.call(this);

```

该句话的意思说，调用原型的handle方法，来继续调用其继任者（也就是successor）的handle方法，以下代码表现为：button/dialog/app三个对象定义的handle都会执行。

```

var app = new Handler({

```

```

        handle: function () {
            console.log('app handle');
        }
    }, 3);

    var dialog = new Handler(app, 1);
    dialog.handle = function () {
        console.log('dialog before ...')
        // 这里做具体的处理操作
        Handler.prototype.handle.call(this); //继续往上走
        console.log('dialog after ...')
    };

    var button = new Handler(dialog, 2);
    button.handle = function () {
        console.log('button before ...')
        // 这里做具体的处理操作
        Handler.prototype.handle.call(this);
        console.log('button after ...')
    };

    button.handle();

```

通过代码的运行结果我们可以看出，如果想先自身处理，然后再调用继任者处理的话，就在末尾执行`Handler.prototype.handle.call(this);`代码，如果想先处理继任者的代码，就在开头执行`Handler.prototype.handle.call(this);`代码。

总结

职责链模式经常和组合模式一起使用，这样一个构件的父构件可以作为其继任者。

同时，DOM里的事件冒泡机制也和此好像有点类似，比如点击一个按钮以后，如果不阻止冒泡，其click事件将一直向父元素冒泡，利用这个机制也可以处理很多相关的问题，比如本系列设计模式享元模式里的《例1：事件集中管理》的示例代码。

参考代码：<https://gist.github.com/1174982>

(39) 设计模式之适配器模式

介绍

适配器模式 (Adapter) 是将一个类 (对象) 的接口 (方法或属性) 转化成客户希望的另外一个接口 (方法或属性) , 适配器模式使得原本由于接口不兼容而不能一起工作的那些类 (对象) 可以一些工作。速成包装器 (wrapper) 。

正文

我们来举一个例子, 鸭子 (Duck) 有飞 (fly) 和嘎嘎叫 (quack) 的行为, 而火鸡虽然也有飞 (fly) 的行为, 但是其叫声是咯咯的 (gobble) 。如果你非要火鸡也要实现嘎嘎叫 (quack) 这个动作, 那我们可以复用鸭子的quack方法, 但是具体的叫还应该是咯咯的, 此时, 我们就可以创建一个火鸡的适配器, 以便让火鸡也支持quack方法, 其内部还是要调用gobble。

OK, 我们开始一步一步实现, 首先要先定义鸭子和火鸡的抽象行为, 也就是各自的方法函数:

```
//鸭子
var Duck = function(){

};
Duck.prototype.fly = function(){
throw new Error("该方法必须被重写!");
};
Duck.prototype.quack = function(){
throw new Error("该方法必须被重写!");
}

//火鸡
var Turkey = function(){

};
Turkey.prototype.fly = function(){
throw new Error(" 该方法必须被重写 !");
};
Turkey.prototype.gobble = function(){
throw new Error(" 该方法必须被重写 !");
};
```

然后再定义具体的鸭子和火鸡的构造函数, 分别为:

```
//鸭子
```

```

var MallardDuck = function () {
    Duck.apply(this);
};
MallardDuck.prototype = new Duck(); //原型是Duck
MallardDuck.prototype.fly = function () {
    console.log("可以飞翔很长的距离!");
};
MallardDuck.prototype.quack = function () {
    console.log("嘎嘎！嘎嘎！");
};

//火鸡
var WildTurkey = function () {
    Turkey.apply(this);
};
WildTurkey.prototype = new Turkey(); //原型是Turkey
WildTurkey.prototype.fly = function () {
    console.log("飞翔的距离貌似有点短!");
};
WildTurkey.prototype.gobble = function () {
    console.log("咯咯！咯咯！");
};

```

为了让火鸡也支持quack方法，我们创建了一个新的火鸡适配器TurkeyAdapter：

```

var TurkeyAdapter = function(oTurkey){
    Duck.apply(this);
    this.oTurkey = oTurkey;
};
TurkeyAdapter.prototype = new Duck();
TurkeyAdapter.prototype.quack = function(){
    this.oTurkey.gobble();
};
TurkeyAdapter.prototype.fly = function(){
    var nFly = 0;
    var nLenFly = 5;
    for(; nFly < nLenFly;){
        this.oTurkey.fly();
        nFly = nFly + 1;
    }
};

```

该构造函数接受一个火鸡的实例对象，然后使用Duck进行apply，其适配器原型是Duck，然后要重新修改其原型的quack方法，以便内部调用oTurkey.gobble()方法。其fly方法也做了一些改变，让火鸡连续飞5次（内部也是调用自身的oTurkey.fly()方法）。

调用方法，就很明了了，测试一下便可以知道结果了：

```

var oMallardDuck = new MallardDuck();

```

```
var oWildTurkey = new WildTurkey();
var oTurkeyAdapter = new TurkeyAdapter(oWildTurkey);

//原有的鸭子行为
oMallardDuck.fly();
oMallardDuck.quack();

//原有的火鸡行为
oWildTurkey.fly();
oWildTurkey.gobble();

//适配器火鸡的行为（火鸡调用鸭子的方法名称）
oTurkeyAdapter.fly();
oTurkeyAdapter.quack();
```

总结

那合适使用适配器模式好呢？如果有以下情况出现时，建议使用：

1. 使用一个已经存在的对象，但其方法或属性接口不符合你的要求；
2. 你想创建一个可复用的对象，该对象可以与其它不相关的对象或不可见对象（即接口方法或属性不兼容的对象）协同工作；
3. 想使用已经存在的对象，但是不能对每一个都进行原型继承以匹配它的接口。对象适配器可以适配它的父对象接口方法或属性。

另外，适配器模式和其它几个模式可能容易让人迷惑，这里说一下大概的区别：

1. 适配器和桥接模式虽然类似，但桥接的出发点不同，桥接的目的是将接口部分和实现部分分离，从而对他们可以更为容易也相对独立的加以改变。而适配器则意味着改变一个已有对象的接口。
2. 装饰者模式增强了其它对象的功能而同时又不改变它的接口，因此它对应程序的透明性比适配器要好，其结果是装饰者支持递归组合，而纯粹使用适配器则是不可能的。
3. 代理模式在不改变它的接口的条件下，为另外一个对象定义了一个代理。

参考：<https://github.com/tcorral/Design-Patterns-in-Javascript/blob/master/Adapter/index.html>

(40) 设计模式之组合模式

介绍

组合模式 (Composite) 将对象组合成树形结构以表示 “部分-整体” 的层次结构，组合模式使得用户对单个对象和组合对象的使用具有一致性。

常见的场景有asp.net里的控件机制 (即control里可以包含子control，可以递归操作、添加、删除子control)，类似的还有DOM的机制，一个DOM节点可以包含子节点，不管是父节点还是子节点都有添加、删除、遍历子节点的通用功能。所以说组合模式的关键是要有一个抽象类，它既可以表示子元素，又可以表示父元素。

正文

举个例子，有家餐厅提供了各种各样的菜品，每个餐桌都有一本菜单，菜单上列出了该餐厅所偶的菜品，有早餐糕点、午餐、晚餐等等，每个餐都有各种各样的菜单项，假设不管是菜单项还是整个菜单都应该是可以打印的，而且可以添加子项，比如午餐可以添加新菜品，而菜单项咖啡也可以添加糖啊什么的。

这种情况，我们就可以利用组合的方式将这些内容表示为层次结构了。我们来逐一分解一下我们的实现步骤。

第一步，先实现我们的 “抽象类” 函数MenuComponent：

```
var MenuComponent = function () {  
};  
MenuComponent.prototype.getName = function () {  
    throw new Error("该方法必须重写!");  
};  
MenuComponent.prototype.getDescription = function () {  
    throw new Error("该方法必须重写!");  
};  
MenuComponent.prototype.getPrice = function () {  
    throw new Error("该方法必须重写!");  
};  
MenuComponent.prototype.isVegetarian = function () {  
    throw new Error("该方法必须重写!");  
};  
MenuComponent.prototype.print = function () {  
    throw new Error("该方法必须重写!");  
};  
MenuComponent.prototype.add = function () {  
    throw new Error("该方法必须重写!");  
};  
MenuComponent.prototype.remove = function () {
```



```

        throw new Error("该方法必须重写!");
    };
    MenuComponent.prototype.getChild = function () {
        throw new Error("该方法必须重写!");
    };
};

```

该函数提供了2种类型的方法，一种是获取信息的，比如价格，名称等，另外一种是通用操作方法，比如打印、添加、删除、获取子菜单。

第二步，创建基本的菜品项：

```

var MenuItem = function (sName, sDescription, bVegetarian, nPrice) {
    MenuComponent.apply(this);
    this.sName = sName;
    this.sDescription = sDescription;
    this.bVegetarian = bVegetarian;
    this.nPrice = nPrice;
};
MenuItem.prototype = new MenuComponent();
MenuItem.prototype.getName = function () {
    return this.sName;
};
MenuItem.prototype.getDescription = function () {
    return this.sDescription;
};
MenuItem.prototype.getPrice = function () {
    return this.nPrice;
};
MenuItem.prototype.isVegetarian = function () {
    return this.bVegetarian;
};
MenuItem.prototype.print = function () {
    console.log(this.getName() + ": " + this.getDescription() + ", " + this.getPrice() + "euros");
};

```

由代码可以看出，我们只重新了原型的4个获取信息的方法和print方法，没有重载其它3个操作方法，因为基本菜品不包含添加、删除、获取子菜品的方式。

第三步，创建菜品：

```

var Menu = function (sName, sDescription) {
    MenuComponent.apply(this);
    this.aMenuComponents = [];
    this.sName = sName;
    this.sDescription = sDescription;
    this.createIterator = function () {
        throw new Error("This method must be overwritten!");
    };
};

```

```

};
Menu.prototype = new MenuComponent();
Menu.prototype.add = function (oMenuComponent) {
    // 添加子菜品
    this.aMenuComponents.push(oMenuComponent);
};
Menu.prototype.remove = function (oMenuComponent) {
    // 删除子菜品
    var aMenuItems = [];
    var nMenuItem = 0;
    var nLenMenuItems = this.aMenuComponents.length;
    var oItem = null;

    for (; nMenuItem < nLenMenuItems; ) {
        oItem = this.aMenuComponents[nMenuItem];
        if (oItem !== oMenuComponent) {
            aMenuItems.push(oItem);
        }
        nMenuItem = nMenuItem + 1;
    }
    this.aMenuComponents = aMenuItems;
};
Menu.prototype.getChild = function (nIndex) {
    // 获取指定的子菜品
    return this.aMenuComponents[nIndex];
};
Menu.prototype.getName = function () {
    return this.sName;
};
Menu.prototype.getDescription = function () {
    return this.sDescription;
};
Menu.prototype.print = function () {
    // 打印当前菜品以及所有的子菜品
    console.log(this.getName() + ": " + this.getDescription());
    console.log("-----");

    var nMenuComponent = 0;
    var nLenMenuComponents = this.aMenuComponents.length;
    var oMenuComponent = null;

    for (; nMenuComponent < nLenMenuComponents; ) {
        oMenuComponent = this.aMenuComponents[nMenuComponent];
        oMenuComponent.print();
        nMenuComponent = nMenuComponent + 1;
    }
};

```

注意上述代码，除了实现了添加、删除、获取方法外，打印print方法是首先打印当前菜品信息，然后循环遍历打印所有子菜品信息。

第四步，创建指定的菜品：

我们可以创建几个真实的菜品，比如晚餐、咖啡、糕点等等，其都是用Menu作为其原型，代码如下：

```
var DinnerMenu = function () {
    Menu.apply(this);
};
DinnerMenu.prototype = new Menu();

var CafeMenu = function () {
    Menu.apply(this);
};
CafeMenu.prototype = new Menu();

var PancakeHouseMenu = function () {
    Menu.apply(this);
};
PancakeHouseMenu.prototype = new Menu();
```

第五步，创建最顶级的菜单容器——菜单本：

```
var Mattress = function (aMenus) {
    this.aMenus = aMenus;
};
Mattress.prototype.printMenu = function () {
    this.aMenus.print();
};
```

该函数接收一个菜单数组作为参数，并且值提供了printMenu方法用于打印所有的菜单内容。

第六步，调用方式：

```
var oPanCakeHouseMenu = new Menu("Pancake House Menu", "Breakfast");
var oDinnerMenu = new Menu("Dinner Menu", "Lunch");
var oCoffeeMenu = new Menu("Cafe Menu", "Dinner");
var oAllMenus = new Menu("ALL MENUS", "All menus combined");

oAllMenus.add(oPanCakeHouseMenu);
oAllMenus.add(oDinnerMenu);

oDinnerMenu.add(new MenuItem("Pasta", "Spaghetti with Marinara Sauce, and
    a slice of sourdough bread", true, 3.89));
oDinnerMenu.add(oCoffeeMenu);

oCoffeeMenu.add(new MenuItem("Express", "Coffee from machine", false, 0.9
    9));

var oMattress = new Mattress(oAllMenus);
```

```
console.log("-----");  
oMattress.printMenu();  
console.log("-----");
```

熟悉asp.net控件开发的同学，是不是看起来很熟悉？

总结

组合模式的使用场景非常明确：

1. 你想表示对象的部分-整体层次结构时；
2. 你希望用户忽略组合对象和单个对象的不同，用户将统一地使用组合结构中的所有对象（方法）

另外该模式经常和装饰者一起使用，它们通常有一个公共的父类（也就是原型），因此装饰必须支持具有add、remove、getChild操作的 component接口。

参考：<https://github.com/tcorral/Design-Patterns-in-Javascript/blob/master/Composite/index.html>

(41) 设计模式之模板方法

介绍

模板方法 (TemplateMethod) 定义了一个操作中的算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

模板方法是一种代码复用的基本技术，在类库中尤为重要，因为他们提取了类库中的公共行为。模板方法导致一种反向的控制结构，这种结构就是传说中的“好莱坞法则”，即“别找我们，我们找你”，这指的是父类调用一个类的操作，而不是相反。具体体现是面向对象编程编程语言里的抽象类（以及其中的抽象方法），以及继承该抽象类（和抽象方法）的子类。

正文

举个例子，泡茶和泡咖啡有同样的步骤，比如烧开水 (boilWater)、冲泡 (brew)、倒在杯子里 (pourOnCup)，加小料 (addCondiments) 等等。但每种饮料冲泡的方法以及所加的小料不一样，所以我们可以利用模板方法实现这个主要步骤。

首先先来定义抽象步骤：

```
var CaffeineBeverage = function () {
};
CaffeineBeverage.prototype.prepareRecipe = function () {
    this.boilWater();
    this.brew();
    this.pourOnCup();
    if (this.customerWantsCondiments()) {
        // 如果可以想加小料，就加上
    }
    this.addCondiments();
};
CaffeineBeverage.prototype.boilWater = function () {
    console.log("将水烧开!");
};
CaffeineBeverage.prototype.pourOnCup = function () {
    console.log("将饮料到再杯子里!");
};
CaffeineBeverage.prototype.brew = function () {
    throw new Error("该方法必须重写!");
};
CaffeineBeverage.prototype.addCondiments = function () {
    throw new Error("该方法必须重写!");
};
// 默认加上小料
```

```
CaffeineBeverage.prototype.customerWantsCondiments = function () {
    return true;
};
```

该函数在原型上扩展了所有的基础步骤，以及主要步骤，冲泡和加小料步骤没有实现，供具体饮料所对应的函数来实现，另外是否加小料（customerWantsCondiments）默认返回true，子函数重写的时候可以重写该值。

下面两个函数分别是冲咖啡和冲茶所对应的函数：

```
// 冲咖啡
var Coffee = function () {
    CaffeineBeverage.apply(this);
};
Coffee.prototype = new CaffeineBeverage();
Coffee.prototype.brew = function () {
    console.log("从咖啡机想咖啡倒进去!");
};
Coffee.prototype.addCondiments = function () {
    console.log("添加糖和牛奶");
};
Coffee.prototype.customerWantsCondiments = function () {
    return confirm("你想添加糖和牛奶吗?");
};

//冲茶叶
var Tea = function () {
    CaffeineBeverage.apply(this);
};
Tea.prototype = new CaffeineBeverage();
Tea.prototype.brew = function () {
    console.log("泡茶叶!");
};
Tea.prototype.addCondiments = function () {
    console.log("添加柠檬!");
};
Tea.prototype.customerWantsCondiments = function () {
    return confirm("你想添加柠檬嘛?");
};
```

另外使用confirm，可以让用户自己选择加不加小料，很不错，不是嘛？

总结

模板方法应用于下列情况：

1. 一次性实现一个算法的不变的部分，并将可变的行为留给子类来实现
2. 各子类中公共的行为应被提取出来并集中到一个公共父类中的避免代码重复，不同之处

分离为新的操作，最后，用一个钓鱼这些新操作的模板方法来替换这些不同的代码

3. 控制子类扩展，模板方法只在特定点调用 “hook” 操作，这样就允许在这些点进行扩展和策略模式不同，模板方法使用继承来改变算法的一部分，而策略模式使用委托来改变整个算法。

参考：<https://github.com/tcorral/Design-Patterns-in-Javascript/blob/master/Template/withHook/index.html>

(42) 设计模式之原型模式

介绍

原型模式 (prototype) 是指用原型实例指向创建对象的种类，并且通过拷贝这些原型创建新的对象。

正文

对于原型模式，我们可以利用JavaScript特有的原型继承特性去创建对象的方式，也就是创建的一个对象作为另外一个对象的prototype属性值。原型对象本身就是有效地利用了每个构造器创建的对象，例如，如果一个构造函数的原型包含了一个name属性（见后面的例子），那通过这个构造函数创建的对象都会有这个属性。

在现有的文献里查看原型模式的定义，没有针对JavaScript的，你可能发现很多讲解的都是关于类的，但是现实情况是基于原型继承的JavaScript完全避免了类 (class) 的概念。我们只是简单从现有的对象进行拷贝来创建对象。

真正的原型继承是作为最新版的ECMAScript5标准提出的，使用Object.create方法来创建这样的对象，该方法创建指定的对象，其对象的prototype有指定的对象（也就是该方法传进的第一个参数对象），也可以包含其他可选的指定属性。例如Object.create(prototype, optionalDescriptorObjects)，下面的例子里也可以看到这个用法：

```
// 因为不是构造函数，所以不用大写
var someCar = {
    drive: function () { },
    name: '马自达 3'
};

// 使用Object.create创建一个新车x
var anotherCar = Object.create(someCar);
anotherCar.name = '丰田佳美';
```

Object.create运行你直接从其它对象继承过来，使用该方法的第二个参数，你可以初始化额外的其它属性。例如：

```
var vehicle = {
    getModel: function () {
        console.log('车辆的模具是：' + this.model);
    }
};
```



```
var car = Object.create(vehicle, {
  'id': {
    value: MY_GLOBAL.nextId(),
    enumerable: true // 默认writable:false, configurable:false
  },
  'model': {
    value: '福特',
    enumerable: true
  }
});
```

这里，可以在Object.create的第二个参数里使用对象字面量传入要初始化的额外属性，其语法与Object.defineProperties或Object.defineProperty方法类型。它允许您设定属性的特性，例如enumerable, writable 或 configurable。

如果你希望自己去实现原型模式，而不直接使用Object.create。你可以使用像下面这样的代码为上面的例子来实现：

```
var vehiclePrototype = {
  init: function (carModel) {
    this.model = carModel;
  },
  getModel: function () {
    console.log('车辆模具是：' + this.model);
  }
};

function vehicle(model) {
  function F() { };
  F.prototype = vehiclePrototype;

  var f = new F();

  f.init(model);
  return f;
}

var car = vehicle('福特Escort');
car.getModel();
```

总结

原型模式在JavaScript里的使用简直是无处不在，其它很多模式有很多也是基于prototype的，就不多说了，这里大家要注意的依然是浅拷贝和深拷贝的问题，免得出引用问题。

(43) 设计模式之状态模式

介绍

状态模式 (State) 允许一个对象在其内部状态改变的时候改变它的行为，对象看起来似乎修改了它的类。

正文

举个例子，就比如我们平时在下载东西，通常就会有好几个状态，比如准备状态 (ReadyState)、下载状态 (DownloadingState)、暂停状态 (DownloadPausedState)、下载完毕状态 (DownloadedState)、失败状态 (DownloadFailedState)，也就是说在每个状态都只可以做当前状态才可以做的事情，而不能做其它状态能做的事儿。

由于State模式描述了下载 (Download) 如何在每一种状态下表现出不同的行为。这一模式的关键思想就是引入了一个叫做State的抽象类 (或JS里的函数) 来表示下载状态，State函数 (作为原型) 为每个状态的子类 (继承函数) 声明了一些公共接口。其每个继承函数实现与特定状态相关的行为，比如DownloadingState和DownloadedState分别实现了正在下载和下载完毕的行为。这些行为可以通过Download来来维护。

让我们来实现一把，首先定义作为其他基础函数的原型的State函数：

```
var State = function () {  
  };  
  
State.prototype.download = function () {  
  throw new Error("该方法必须被重载!");  
};  
  
State.prototype.pause = function () {  
  throw new Error("该方法必须被重载!");  
};  
  
State.prototype.fail = function () {  
  throw new Error("该方法必须被重载!");  
};  
  
State.prototype.finish = function () {  
  throw new Error("该方法必须被重载!");  
};
```

我们为State的原型定义了4个方法接口，分别对应着下载（download）、暂停（pause）、失败（fail）、结束（finish）以便子函数可以重写。

在编写子函数之前，我们先来编写一个ReadyState函数，以便可以将状态传递给第一个download状态：

```
var ReadyState = function (oDownload) {
    State.apply(this);
    this.oDownload = oDownload;
};

ReadyState.prototype = new State();

ReadyState.prototype.download = function () {
    this.oDownload.setState(this.oDownload.getDownloadingState());
    // Ready以后，可以开始下载，所以设置了Download函数里的状态获取方法
    console.log("Start Download!");
};

ReadyState.prototype.pause = function () {
    throw new Error("还没开始下载，不能暂停!");
};

ReadyState.prototype.fail = function () {
    throw new Error("文件还没开始下载，怎么能说失败呢!");
};

ReadyState.prototype.finish = function () {
    throw new Error("文件还没开始下载，当然也不能结束了!");
};
```

该函数接收了一个Download维护函数的实例作为参数，Download函数用于控制状态的改变和获取（类似于中央控制器，让外部调用），ReadyState重写了原型的download方法，以便开始进行下载。我们继续来看Download函数的主要功能：

```
var Download = function () {
    this.oState = new ReadyState(this);
};

Download.prototype.setState = function (oState) {
    this.oState = oState;
};

// 对外暴露的四个公共方法，以便外部调用

Download.prototype.download = function () {
    this.oState.download();
};
```

```

Download.prototype.pause = function () {
    this.oState.pause();
};

Download.prototype.fail = function () {
    this.oState.fail();
};

Download.prototype.finish = function () {
    this.oState.finish();
};

//获取各种状态, 传入当前this对象
Download.prototype.getReadyState = function () {
    return new ReadyState(this);
};

Download.prototype.getDownloadingState = function () {
    return new DownloadingState(this);
};

Download.prototype.getDownloadPausedState = function () {
    return new DownloadPausedState(this);
};

Download.prototype.getDownloadedState = function () {
    return new DownloadedState(this);
};

Download.prototype.getDownloadedFailedState = function () {
    return new DownloadFailedState(this);
};

```

Download函数的原型提供了8个方法, 4个是对用于下载状态的操作行为, 另外4个是用于获取当前四个不同的状态, 这4个方法都接收this作为参数, 也就是将Download实例自身作为一个参数传递给处理该请求的状态对象 (ReadyState 以及后面要实现的继承函数), 这使得状态对象比必要的时候可以访问oDownload。

接下来, 继续定义4个相关状态的函数:

```

var DownloadingState = function (oDownload) {
    State.apply(this);
    this.oDownload = oDownload;
};

DownloadingState.prototype = new State();

DownloadingState.prototype.download = function () {
    throw new Error("文件已经正在下载中了!");
};

```

```

DownloadingState.prototype.pause = function () { this.oDownload.setState(
this.oDownload.getDownloadPausedState());
    console.log("暂停下载!");
};

DownloadingState.prototype.fail = function () { this.oDownload.setState(t
his.oDownload.getDownloadFailedState());
    console.log("下载失败!");
};

DownloadingState.prototype.finish = function () {
    this.oDownload.setState(this.oDownload.getDownloadedState());
    console.log("下载完毕!");
};

```

DownloadingState的主要注意事项就是已经正在下载的文件，不能再次开始下载了，其它的状态都可以连续进行。

```

var DownloadPausedState = function (oDownload) {
    State.apply(this);
    this.oDownload = oDownload;
};

DownloadPausedState.prototype = new State();

DownloadPausedState.prototype.download = function () {
    this.oDownload.setState(this.oDownload.getDownloadingState());
    console.log("继续下载!");
};

DownloadPausedState.prototype.pause = function () {
    throw new Error("已经暂停了，咋还要暂停呢!");
};

DownloadPausedState.prototype.fail = function () { this.oDownload.setStat
e(this.oDownload.getDownloadFailedState());
    console.log("下载失败!");
};

DownloadPausedState.prototype.finish = function () {
    this.oDownload.setState(this.oDownload.getDownloadedState());
    console.log("下载完毕!");
};

```

DownloadPausedState函数里要注意的是，已经暂停的下载，不能再次暂停。

```

var DownloadedState = function (oDownload) {
    State.apply(this);
    this.oDownload = oDownload;
};

```

```

DownloadedState.prototype = new State();

DownloadedState.prototype.download = function () {
    this.oDownload.setState(this.oDownload.getDownloadingState());
    console.log("重新下载!");
};

DownloadedState.prototype.pause = function () {
    throw new Error("对下载完了, 还暂停啥?");
};

DownloadedState.prototype.fail = function () {
    throw new Error("都下载成功了, 咋会失败呢?");
};

DownloadedState.prototype.finish = function () {
    throw new Error("下载成功了, 不能再为成功了吧!");
};

```

DownloadedState函数, 同理成功下载以后, 不能再设置finish了, 只能设置重新下载状态。

```

var DownloadFailedState = function (oDownload) {
    State.apply(this);
    this.oDownload = oDownload;
};

DownloadFailedState.prototype = new State();

DownloadFailedState.prototype.download = function () {
    this.oDownload.setState(this.oDownload.getDownloadingState());
    console.log("尝试重新下载!");
};

DownloadFailedState.prototype.pause = function () {
    throw new Error("失败的下载, 也不能暂停!");
};

DownloadFailedState.prototype.fail = function () {
    throw new Error("都失败了, 咋还失败呢!");
};

DownloadFailedState.prototype.finish = function () {
    throw new Error("失败的下载, 肯定也不会成功!");
};

```

同理, DownloadFailedState函数的失败状态, 也不能再次失败, 但可以和finished以后再次尝试重新下载。

调用测试代码, 就非常简单了, 我们在HTML里演示吧, 首先是要了jquery, 然后有3个按钮分别代表: 开始下载、暂停、重新下载。(注意在Firefox里用firebug查看结果, 因为用了

console.log方法)。

```
<html>
<head>
  <link type="text/css" rel="stylesheet" href="http://www.cnblogs.com/c
ss/style.css" />
  <title>State Pattern</title>
  <script type="text/javascript" src="/jquery.js"></script>
  <script type="text/javascript" src="Download.js"></script>
  <script type="text/javascript" src="states/State.js"></script>
  <script type="text/javascript" src="states/DownloadFailedState.js"></
script>
  <script type="text/javascript" src="states/DownloadPausedState.js"></
script>
  <script type="text/javascript" src="states/DownloadedState.js"></scri
pt>
  <script type="text/javascript" src="states/DownloadingState.js"></scr
ipt>
  <script type="text/javascript" src="states/ReadyState.js"></script>
</head>
<body>
  <input type="button" value="开始下载" id="download_button" />
  <input type="button" value="暂停" id="pause_button" />
  <input type="button" value="重新下载" id="resume_button" />
  <script type="text/javascript">
    var oDownload = new Download();
    $("#download_button").click(function () {
      oDownload.download();
    });

    $("#pause_button").click(function () {
      oDownload.pause();
    });

    $("#resume_button").click(function () {
      oDownload.download();
    });
  </script>
</body>
</html>
```

总结

状态模式的使用场景也特别明确，有如下两点：

1. 一个对象的行为取决于它的状态，并且它必须在运行时刻根据状态改变它的行为。
2. 一个操作中含有大量的分支语句，而且这些分支语句依赖于该对象的状态。状态通常为一个或多个枚举常量的表示。

参考：<https://github.com/tcorral/Design-Patterns-in->

[Javascript/blob/master/State/1/index.html](#)

(44) 设计模式之桥接模式

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;">Connectio
n Error!</span>';
            });
            // 订阅全部成功的处理步骤x
            q.onComplete.subscribe(function () {
                results.innerHTML += ' <span style="color:green;">Comple

```



```

ed!</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=' + dat
a,
            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>

```

```
</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。

(45) 代码复用模式 (避免篇)

介绍

任何编程都提出代码复用，否则每次开发一个新程序或者写一个新功能都要全新编写的话，那就歇菜了，但是代码复用也是有好要坏，接下来的两篇文章我们将针对代码复用来进行讨论，第一篇文避免篇，指的是要尽量避免使用这些模式，因为或多或少有带来一些问题；第二排是推荐篇，指的是推荐大家使用的模式，一般不会有什么问题。

模式1：默认模式

代码复用大家常用的默认模式，往往是有问题的，该模式使用Parent()的构造函数创建一个对象，并且将该对象赋值给Child()的原型。我们看一下代码：

```
function inherit(C, P) {
    C.prototype = new P();
}

// 父构造函数
function Parent(name) {
    this.name = name || 'Adam';
}
// 给原型添加say功能
Parent.prototype.say = function () {
    return this.name;
};
// Child构造函数为空
function Child(name) {
}

// 执行继承
inherit(Child, Parent);

var kid = new Child();
console.log(kid.say()); // "Adam"

var kiddo = new Child();
kiddo.name = "Patrick";
console.log(kiddo.say()); // "Patrick"

// 缺点:不能让参数传进给Child构造函数
var s = new Child('Seth');
console.log(s.say()); // "Adam"
```

这种模式的缺点是Child不能传进参数，基本上也就废了。

模式2：借用构造函数

本文档使用 [看云](#) 构建

该模式是Child借用Parent的构造函数进行apply，然后将child的this和参数传递给apply方法：

```
// 父构造函数
function Parent(name) {
    this.name = name || 'Adam';
}

// 给原型添加say功能
Parent.prototype.say = function () {
    return this.name;
};

// Child构造函数
function Child(name) {
    Parent.apply(this, arguments);
}

var kid = new Child("Patrick");
console.log(kid.name); // "Patrick"

// 缺点：没有从构造函数上继承say方法
console.log(typeof kid.say); // "undefined"
```

缺点也很明显，say方法不可用，因为没有继承过来。

模式3：借用构造函数并设置原型

上述两个模式都有自己的缺点，那如何把两者的缺点去除呢，我们来尝试一下：

```
// 父构造函数
function Parent(name) {
    this.name = name || 'Adam';
}

// 给原型添加say功能
Parent.prototype.say = function () {
    return this.name;
};

// Child构造函数
function Child(name) {
    Parent.apply(this, arguments);
}

Child.prototype = new Parent();

var kid = new Child("Patrick");
console.log(kid.name); // "Patrick"
console.log(typeof kid.say); // function
console.log(kid.say()); // Patrick
```

```
console.dir(kid);
delete kid.name;
console.log(kid.say()); // "Adam"
```

运行起来，一切正常，但是有没有发现，Parent构造函数执行了两次，所以说，虽然程序可用，但是效率很低。

模式4：共享原型

共享原型是指Child和Parent使用同样的原型，代码如下：

```
function inherit(C, P) {
    C.prototype = P.prototype;
}

// 父构造函数
function Parent(name) {
    this.name = name || 'Adam';
}

// 给原型添加say功能
Parent.prototype.say = function () {
    return this.name;
};

// Child构造函数
function Child(name) {
}

inherit(Child, Parent);

var kid = new Child('Patrick');
console.log(kid.name); // undefined
console.log(typeof kid.say); // function
kid.name = 'Patrick';
console.log(kid.say()); // Patrick
console.dir(kid);
```

确定还是一样，Child的参数没有正确接收到。

模式5：临时构造函数

首先借用构造函数，然后将Child的原型设置为该借用构造函数的实例，最后恢复Child原型的构造函数。代码如下：

```
/* 闭包 */
var inherit = (function () {
    var F = function () {
    };

```

```

        return function (C, P) {
            F.prototype = P.prototype;
            C.prototype = new F();
            C.uber = P.prototype;
            C.prototype.constructor = C;
        }
    } ());

function Parent(name) {
    this.name = name || 'Adam';
}

// 给原型添加say功能
Parent.prototype.say = function () {
    return this.name;
};

// Child构造函数
function Child(name) {
}

inherit(Child, Parent);

var kid = new Child();
console.log(kid.name); // undefined
console.log(typeof kid.say); // function
kid.name = 'Patrick';
console.log(kid.say()); // Patrick
var kid2 = new Child("Tom");
console.log(kid.say());
console.log(kid.constructor.name); // Child
console.log(kid.constructor === Parent); // false

```

问题照旧，Child不能正常接收参数。

模式6：klass

这个模式，先上代码吧：

```

var klass = function (Parent, props) {

    var Child, F, i;

    // 1.
    // 新构造函数
    Child = function () {
        if (Child.uber && Child.uber.hasOwnProperty("__construct")) {
            Child.uber.__construct.apply(this, arguments);
        }
        if (Child.prototype.hasOwnProperty("__construct")) {
            Child.prototype.__construct.apply(this, arguments);
        }
    }

};

```

```

    // 2.
    // 继承
    Parent = Parent || Object;
    F = function () {
    };
    F.prototype = Parent.prototype;
    Child.prototype = new F();
    Child.uber = Parent.prototype;
    Child.prototype.constructor = Child;

    // 3.
    // 添加实现方法
    for (i in props) {
        if (props.hasOwnProperty(i)) {
            Child.prototype[i] = props[i];
        }
    }

    // return the "class"
    return Child;
};

var Man = klass(null, {
    __construct: function (what) {
        console.log("Man's constructor");
        this.name = what;
    },
    getName: function () {
        return this.name;
    }
});

var first = new Man('Adam'); // logs "Man's constructor"
first.getName(); // "Adam"

var SuperMan = klass(Man, {
    __construct: function (what) {
        console.log("SuperMan's constructor");
    },
    getName: function () {
        var name = SuperMan.uber.getName.call(this);
        return "I am " + name;
    }
});

var clark = new SuperMan('Clark Kent');
clark.getName(); // "I am Clark Kent"

console.log(clark instanceof Man); // true
console.log(clark instanceof SuperMan); // true

```

怎么样？看着是不是有点晕，说好点，该模式的语法和规范拧得和别的语言一样，你愿意用么？咳。。。

总结

以上六个模式虽然在某种特殊情况下实现了某些功能，但是都存在各自的缺点，所以一般情况下，大家要避免使用。

参考：<http://shichuan.github.com/javascript-patterns/#code-reuse-patterns>

(46) 代码复用模式 (推荐篇)

介绍

本文介绍的四种代码复用模式都是最佳实践，推荐大家在编程的过程中使用。

模式1：原型继承

原型继承是让父对象作为子对象的原型，从而达到继承的目的：

```
function object(o) {
    function F() {
    }

    F.prototype = o;
    return new F();
}

// 要继承的父对象
var parent = {
    name: "Papa"
};

// 新对象
var child = object(parent);

// 测试
console.log(child.name); // "Papa"

// 父构造函数
function Person() {
    // an "own" property
    this.name = "Adam";
}
// 给原型添加新属性
Person.prototype.getName = function () {
    return this.name;
};
// 创建新person
var papa = new Person();
// 继承
var kid = object(papa);
console.log(kid.getName()); // "Adam"

// 父构造函数
function Person() {
    // an "own" property
    this.name = "Adam";
}
```

```
// 给原型添加新属性
Person.prototype.getName = function () {
    return this.name;
};
// 继承
var kid = object(Person.prototype);
console.log(typeof kid.getName); // "function", 因为是在原型里定义的
console.log(typeof kid.name); // "undefined", 因为只继承了原型
```

同时，ECMAScript5也提供了类似的一个方法叫做Object.create用于继承对象，用法如下：

```
/* 使用新版的ECMAScript 5提供的功能 */
var child = Object.create(parent);

var child = Object.create(parent, {
    age: { value: 2 } // ECMA5 descriptor
});
console.log(child.hasOwnProperty("age")); // true
```

而且，也可以更细粒度地在第二个参数上定义属性：

```
// 首先，定义一个新对象man
var man = Object.create(null);

// 接着，创建包含属性的配置设置
// 属性设置为可写，可枚举，可配置
var config = {
    writable: true,
    enumerable: true,
    configurable: true
};

// 通常使用Object.defineProperty()来添加新属性(ECMAScript5支持)
// 现在，为了方便，我们自定义一个封装函数
var defineProp = function (obj, key, value) {
    config.value = value;
    Object.defineProperty(obj, key, config);
}

defineProp(man, 'car', 'Delorean');
defineProp(man, 'dob', '1981');
defineProp(man, 'beard', false);
```

所以，继承就这么可以做了：

```
var driver = Object.create( man );
defineProp (driver, 'topSpeed', '100mph');
driver.topSpeed // 100mph
```

但是有个地方需要注意，就是Object.create(null)创建的对象的原型为undefined，也就是没有toString和valueOf方法，所以alert(man);的时候会出错，但alert(man.car);是没问题的。

模式2：复制所有属性进行继承

这种方式的继承就是将父对象里所有的属性都复制到子对象上，一般子对象可以使用父对象的数据。

先来看一个浅拷贝的例子：

```
/* 浅拷贝 */
function extend(parent, child) {
    var i;
    child = child || {};
    for (i in parent) {
        if (parent.hasOwnProperty(i)) {
            child[i] = parent[i];
        }
    }
    return child;
}

var dad = { name: "Adam" };
var kid = extend(dad);
console.log(kid.name); // "Adam"

var dad = {
    counts: [1, 2, 3],
    reads: { paper: true }
};
var kid = extend(dad);
kid.counts.push(4);
console.log(dad.counts.toString()); // "1,2,3,4"
console.log(dad.reads === kid.reads); // true
```

代码的最后一行，你可以发现dad和kid的reads是一样的，也就是他们使用的是同一个引用，这也就是浅拷贝带来的问题。

我们再来看一下深拷贝：

```
/* 深拷贝 */
function extendDeep(parent, child) {
    var i,
        toStr = Object.prototype.toString,
        astr = "[object Array]";

    child = child || {};
    for (i in parent) {
        if (parent.hasOwnProperty(i)) {
            if (toStr.call(parent[i]) === astr) {
                child[i] = extendDeep(parent[i], child[i]);
            } else {
                child[i] = parent[i];
            }
        }
    }
    return child;
}
```

```

    for (i in parent) {
        if (parent.hasOwnProperty(i)) {
            if (typeof parent[i] === 'object') {
                child[i] = (toStr.call(parent[i]) === astr) ? [] : {};
                extendDeep(parent[i], child[i]);
            } else {
                child[i] = parent[i];
            }
        }
    }
    return child;
}

var dad = {
    counts: [1, 2, 3],
    reads: { paper: true }
};
var kid = extendDeep(dad);

kid.counts.push(4);
console.log(kid.counts.toString()); // "1,2,3,4"
console.log(dad.counts.toString()); // "1,2,3"

console.log(dad.reads === kid.reads); // false
kid.reads.paper = false;

```

深拷贝以后，两个值就不相等了，bingo！

模式3：混合 (mix-in)

混入就是将一个对象的一个或多个（或全部）属性（或方法）复制到另外一个对象，我们举一个例子：

```

function mix() {
    var arg, prop, child = {};
    for (arg = 0; arg < arguments.length; arg += 1) {
        for (prop in arguments[arg]) {
            if (arguments[arg].hasOwnProperty(prop)) {
                child[prop] = arguments[arg][prop];
            }
        }
    }
    return child;
}

var cake = mix(
    { eggs: 2, large: true },
    { butter: 1, salted: true },
    { flour: '3 cups' },
    { sugar: 'sure!' }
);

```

```
console.dir(cake);
```

mix函数将所传入的所有参数的子属性都复制到child对象里，以便产生一个新对象。

那如何我们只想混入部分属性呢？该个如何做？其实我们可以使用多余的参数来定义需要混入的属性，例如mix (child,parent,method1,method2)这样就可以只将parent里的method1和method2混入到child里。上代码：

```
// Car
var Car = function (settings) {
    this.model = settings.model || 'no model provided';
    this.colour = settings.colour || 'no colour provided';
};

// Mixin
var Mixin = function () { };
Mixin.prototype = {
    driveForward: function () {
        console.log('drive forward');
    },
    driveBackward: function () {
        console.log('drive backward');
    }
};

// 定义的2个参数分别是被混入的对象 (receiving) 和从哪里混入的对象 (giving)
function augment(receivingObj, givingObj) {
    // 如果提供了指定的方法名称的话，也就是参数多余3个
    if (arguments[2]) {
        for (var i = 2, len = arguments.length; i < len; i++) {
            receivingObj.prototype[arguments[i]] = givingObj.prototype[arguments[i]];
        }
    }
    // 如果不指定第3个参数，或者更多参数，就混入所有的方法
    else {
        for (var methodName in givingObj.prototype) {
            // 检查receiving对象内部不包含要混入的名字，如何包含就不混入了
            if (!receivingObj.prototype[methodName]) {
                receivingObj.prototype[methodName] = givingObj.prototype[methodName];
            }
        }
    }
}

// 给Car混入属性，但是值混入'driveForward' 和 'driveBackward'*/
augment(Car, Mixin, 'driveForward', 'driveBackward');

// 创建新对象Car
var vehicle = new Car({ model: 'Ford Escort', colour: 'blue' });
```

```
// 测试是否成功得到混入的方法
vehicle.driveForward();
vehicle.driveBackward();
```

该方法使用起来就比较灵活了。

模式4：借用方法

一个对象借用另外一个对象的一个或两个方法，而这两个对象之间不会有什么直接联系。不用多解释，直接用代码解释吧：

```
var one = {
  name: 'object',
  say: function (greet) {
    return greet + ', ' + this.name;
  }
};

// 测试
console.log(one.say('hi')); // "hi, object"

var two = {
  name: 'another object'
};

console.log(one.say.apply(two, ['hello'])); // "hello, another object"

// 将say赋值给一个变量，this将指向到全局变量
var say = one.say;
console.log(say('hoho')); // "hoho, undefined"

// 传入一个回调函数callback
var yetanother = {
  name: 'Yet another object',
  method: function (callback) {
    return callback('Hola');
  }
};
console.log(yetanother.method(one.say)); // "Holla, undefined"

function bind(o, m) {
  return function () {
    return m.apply(o, [].slice.call(arguments));
  };
}

var twosay = bind(two, one.say);
console.log(twosay('yo')); // "yo, another object"

// ECMAScript 5给Function.prototype添加了一个bind()方法，以便很容易使用apply()
和call()。
```

```
if (typeof Function.prototype.bind === 'undefined') {
    Function.prototype.bind = function (thisArg) {
        var fn = this,
            slice = Array.prototype.slice,
            args = slice.call(arguments, 1);
        return function () {
            return fn.apply(thisArg, args.concat(slice.call(arguments)));
        };
    };
}

var twosay2 = one.say.bind(two);
console.log(twosay2('Bonjour')); // "Bonjour, another object"

var twosay3 = one.say.bind(two, 'Enchanté');
console.log(twosay3()); // "Enchanté, another object"
```

总结

就不用总结了吧。

(47) 对象创建模式 (上篇)

介绍

本篇主要是介绍创建对象方面的模式，利用各种技巧可以极大地避免了错误或者可以编写出非常精简的代码。

模式1：命名空间 (namespace)

命名空间可以减少全局命名所需的数量，避免命名冲突或过度。一般我们在进行对象层级定义的时候，经常是这样的：

```
var app = app || {};  
app.moduleA = app.moduleA || {};  
app.moduleA.subModule = app.moduleA.subModule || {};  
app.moduleA.subModule.MethodA = function () {  
    console.log("print A");  
};  
app.moduleA.subModule.MethodB = function () {  
    console.log("print B");  
};
```

如果层级很多的话，那就要一直这样继续下去，很是混乱。namespace模式就是为了解决这个问题而存在的，我们看代码：

```
// 不安全，可能会覆盖已有的MYAPP对象  
var MYAPP = {};  
// 还好  
if (typeof MYAPP === "undefined") {  
    var MYAPP = {};  
}  
// 更简洁的方式  
var MYAPP = MYAPP || {};  
  
//定义通用方法  
MYAPP.namespace = function (ns_string) {  
    var parts = ns_string.split('.'),  
        parent = MYAPP,  
        i;  
  
    // 默认如果第一个节点是MYAPP的话，就忽略掉，比如MYAPP.ModuleA  
    if (parts[0] === "MYAPP") {  
        parts = parts.slice(1);  
    }  
  
    for (i = 0; i < parts.length; i += 1) {  
        // 如果属性不存在，就创建
```

```

        if (typeof parent[parts[i]] === "undefined") {
            parent[parts[i]] = {};
        }
        parent = parent[parts[i]];
    }
    return parent;
};

```

调用代码，非常简单：

```

// 通过namespace以后，可以将返回值赋给一个局部变量
var module2 = MYAPP.namespace('MYAPP.modules.module2');
console.log(module2 === MYAPP.modules.module2); // true

// 跳过MYAPP
MYAPP.namespace('modules.module51');

// 非常长的名字
MYAPP.namespace('once.upon.a.time.there.was.this.long.nested.property');

```

模式2：定义依赖

有时候你的一个模块或者函数可能要引用第三方的一些模块或者工具，这时候最好将这些依赖模块在刚开始的时候就定义好，以便以后可以很方便地替换掉。

```

var myFunction = function () {
    // 依赖模块
    var event = YAHOO.util.Event,
        dom = YAHOO.util.dom;

    // 其它函数后面的代码里使用局部变量event和dom
};

```

模式3：私有属性和私有方法

JavaScript本书不提供特定的语法来支持私有属性和私有方法，但是我们可以通过闭包来实现，代码如下：

```

function Gadget() {
    // 私有对象
    var name = 'iPod';
    // 公有函数
    this.getName = function () {
        return name;
    };
}
var toy = new Gadget();

```

```

// name未定义, 是私有的
console.log(toy.name); // undefined

// 公有方法访问name
console.log(toy.getName()); // "iPod"

var myobj; // 通过自执行函数给myobj赋值
(function () {
    // 自由对象
    var name = "my, oh my";

    // 实现了公有部分, 所以没有var
    myobj = {
        // 授权方法
        getName: function () {
            return name;
        }
    };
} ());

```

模式4：Revelation模式

也是关于隐藏私有方法的模式，和《[深入理解JavaScript系列（3）：全面解析Module模式](#)》里的Module模式有点类似，但是不是return的方式，而是在外部先声明一个变量，然后在内部给变量赋值公有方法。代码如下：

```

var myarray;

(function () {
    var astr = "[object Array]",
        toString = Object.prototype.toString;

    function isArray(a) {
        return toString.call(a) === astr;
    }

    function indexOf(haystack, needle) {
        var i = 0,
            max = haystack.length;
        for (; i < max; i += 1) {
            if (haystack[i] === needle) {
                return i;
            }
        }
        return -1;
    }

    //通过赋值的方式, 将上面所有的细节都隐藏了
    myarray = {
        isArray: isArray,
        indexOf: indexOf,
    }
} ());

```

```

        inArray: indexOf
    };
} ());

//测试代码
console.log(myarray.isArray([1, 2])); // true
console.log(myarray.isArray({ 0: 1 })); // false
console.log(myarray.indexOf(["a", "b", "z"], "z")); // 2
console.log(myarray.inArray(["a", "b", "z"], "z")); // 2

myarray.indexOf = null;
console.log(myarray.inArray(["a", "b", "z"], "z")); // 2

```

模式5：链模式

链模式可以让你连续调用一个对象的方法，比如obj.add(1).remove(2).delete(4).add(2)这样的形式，其实现思路非常简单，就是将this原样返回。代码如下：

```

var obj = {
    value: 1,
    increment: function () {
        this.value += 1;
        return this;
    },
    add: function (v) {
        this.value += v;
        return this;
    },
    shout: function () {
        console.log(this.value);
    }
};

// 链方法调用
obj.increment().add(3).shout(); // 5

// 也可以单独一个一个调用
obj.increment();
obj.add(3);
obj.shout();

```

总结

本篇是对象创建模式的上篇，敬请期待明天的下篇。

参考：<http://shichuan.github.com/javascript-patterns/#object-creation-patterns>

(48) 对象创建模式 (下篇)

介绍

本篇主要是介绍创建对象方面的模式的下篇，利用各种技巧可以极大地避免了错误或者可以编写出非常精简的代码。

模式6：函数语法糖

函数语法糖是为一个对象快速添加方法（函数）的扩展，这个主要是利用prototype的特性，代码比较简单，我们先来看一下实现代码：

```
if (typeof Function.prototype.method !== "function") {  
    Function.prototype.method = function (name, implementation) {  
        this.prototype[name] = implementation;  
        return this;  
    };  
}
```

扩展对象的时候，可以这么用：

```
var Person = function (name) {  
    this.name = name;  
}  
.method('getName',  
        function () {  
            return this.name;  
        })  
.method('setName', function (name) {  
    this.name = name;  
    return this;  
});
```

这样就给Person函数添加了getName和setName这2个方法，接下来我们来验证一下结果：

```
var a = new Person('Adam');  
console.log(a.getName()); // 'Adam'  
console.log(a.setName('Eve').getName()); // 'Eve'
```

模式7：对象常量

对象常量是在一个对象提供set,get,ifDefined各种方法的体现，而且对于set的方法只会保留最先设置的对象，后期再设置都是无效的，已达到别人无法重载的目的。实现代码如下：

```

var constant = (function () {
    var constants = {},
        ownProp = Object.prototype.hasOwnProperty,
        // 只允许设置这三种类型的值
        allowed = {
            string: 1,
            number: 1,
            boolean: 1
        },
        prefix = (Math.random() + "_").slice(2);

    return {
        // 设置名称为name的属性
        set: function (name, value) {
            if (this.isDefined(name)) {
                return false;
            }
            if (!ownProp.call(allowed, typeof value)) {
                return false;
            }
            constants[prefix + name] = value;
            return true;
        },
        // 判断是否存在名称为name的属性
        isDefined: function (name) {
            return ownProp.call(constants, prefix + name);
        },
        // 获取名称为name的属性
        get: function (name) {
            if (this.isDefined(name)) {
                return constants[prefix + name];
            }
            return null;
        }
    };
})();

```

验证代码如下：

```

// 检查是否存在
console.log(constant.isDefined("maxwidth")); // false

// 定义
console.log(constant.set("maxwidth", 480)); // true

// 重新检测
console.log(constant.isDefined("maxwidth")); // true

// 尝试重新定义
console.log(constant.set("maxwidth", 320)); // false

// 判断原先的定义是否还存在

```

```
console.log(constant.get("maxwidth")); // 480
```

模式8：沙盒模式

沙盒 (Sandbox) 模式即时为一个或多个模块提供单独的上下文环境，而不会影响其他模块的上下文环境，比如有个Sandbox里有3个方法event,dom,ajax，在调用其中2个组成一个环境的话，和调用三个组成的环境完全没有干扰。Sandbox实现代码如下：

```
function Sandbox() {
    // 将参数转为数组
    var args = Array.prototype.slice.call(arguments),
    // 最后一个参数为callback
    callback = args.pop(),
    // 除最后一个参数外，其它均为要选择的模块
    modules = (args[0] && typeof args[0] === "string") ? args : args[
0],
    i;

    // 强制使用new操作符
    if (!(this instanceof Sandbox)) {
        return new Sandbox(modules, callback);
    }

    // 添加属性
    this.a = 1;
    this.b = 2;

    // 向this对象上需想添加模块
    // 如果没有模块或传入的参数为 "*"，则以为着传入所有模块
    if (!modules || modules == '*') {
        modules = [];
        for (i in Sandbox.modules) {
            if (Sandbox.modules.hasOwnProperty(i)) {
                modules.push(i);
            }
        }
    }

    // 初始化需要的模块
    for (i = 0; i < modules.length; i += 1) {
        Sandbox.modules[modules[i]](this);
    }

    // 调用 callback
    callback(this);
}

// 默认添加原型对象
Sandbox.prototype = {
    name: "My Application",
    version: "1.0",
    getName: function () {
```



```

        return this.name;
    }
};

```

然后我们再定义默认的初始模块：

```

Sandbox.modules = {};

Sandbox.modules.dom = function (box) {
    box.getElement = function () {
    };
    box.getStyle = function () {
    };
    box.foo = "bar";
};

Sandbox.modules.event = function (box) {
    // access to the Sandbox prototype if needed:
    // box.constructor.prototype.m = "mmm";
    box.attachEvent = function () {
    };
    box.detachEvent = function () {
    };
};

Sandbox.modules.ajax = function (box) {
    box.makeRequest = function () {
    };
    box.getResponse = function () {
    };
};

```

调用方式如下：

```

// 调用方式
Sandbox(['ajax', 'event'], function (box) {
    console.log(typeof (box.foo));
    // 没有选择dom, 所以box.foo不存在
});

Sandbox('ajax', 'dom', function (box) {
    console.log(typeof (box.attachEvent));
    // 没有选择event, 所以event里定义的attachEvent也不存在
});

Sandbox('*', function (box) {
    console.log(box); // 上面定义的所有方法都可访问
});

```

通过三个不同的调用方式，我们可以看到，三种方式的上下文环境都是不同的，第一种里没
本文档使用 [看云](#) 构建

有foo; 而第二种则没有attachEvent, 因为只加载了ajax和dom, 而没有加载event; 第三种则加载了全部。

模式9：静态成员

静态成员 (Static Members) 只是一个函数或对象提供的静态属性, 可分为私有的和公有的, 就像C#或Java里的public static和private static一样。

我们先来看一下公有成员, 公有成员非常简单, 我们平时声明的方法, 函数都是公有的, 比如:

```
// 构造函数
var Gadget = function () {
};

// 公有静态方法
Gadget.isShiny = function () {
    return "you bet";
};

// 原型上添加的正常方法
Gadget.prototype.setPrice = function (price) {
    this.price = price;
};

// 调用静态方法
console.log(Gadget.isShiny()); // "you bet"

// 创建实例, 然后调用方法
var iphone = new Gadget();
iphone.setPrice(500);

console.log(typeof Gadget.setPrice); // "undefined"
console.log(typeof iphone.isShiny); // "undefined"
Gadget.prototype.isShiny = Gadget.isShiny;
console.log(iphone.isShiny()); // "you bet"
```

而私有静态成员, 我们可以利用其闭包特性去实现, 以下是两种实现方式。

第一种实现方式:

```
var Gadget = (function () {
    // 静态变量/属性
    var counter = 0;

    // 闭包返回构造函数的新实现
    return function () {
        console.log(counter += 1);
    };
};
```

```

} ()); // 立即执行

var g1 = new Gadget(); // logs 1
var g2 = new Gadget(); // logs 2
var g3 = new Gadget(); // logs 3

```

可以看出，虽然每次都是new的对象，但数字依然是递增的，达到了静态成员的目的。

第二种方式：

```

var Gadget = (function () {
    // 静态变量/属性
    var counter = 0,
        NewGadget;

    // 新构造函数实现
    NewGadget = function () {
        counter += 1;
    };

    // 授权可以访问的方法
    NewGadget.prototype.getLastId = function () {
        return counter;
    };

    // 覆盖构造函数
    return NewGadget;
} ()); // 立即执行

var iphone = new Gadget();
iphone.getLastId(); // 1
var ipod = new Gadget();
ipod.getLastId(); // 2
var ipad = new Gadget();
ipad.getLastId(); // 3

```

数字也是递增了，这是利用其内部授权方法的闭包特性实现的。

总结

这是对象创建模式的下篇，两篇一起总共9种模式，是我们在日常JavaScript编程中经常使用的对象创建模式，不同的场景起到了不同的作用，希望大家根据各自的需求选择适用的模式。

参考：<http://shichuan.github.com/javascript-patterns/#object-creation-patterns>

(49) Function模式 (上篇)

介绍

本篇主要是介绍Function方面使用的一些技巧 (上篇)，利用Function特性可以编写出很多非常有意思的代码，本篇主要包括：回调模式、配置对象、返回函数、分布程序、柯里化 (Currying)。

回调函数

在JavaScript中，当一个函数A作为另外一个函数B的其中一个参数时，则函数A称为回调函数，即A可以在函数B的周期内执行 (开始、中间、结束时均可)。

举例来说，有一个函数用于生成node

```
var complexComputation = function () { /* 内部处理，并返回一个node*/};
```

有一个findNodes函数声明用于查找所有的节点，然后通过callback回调进行执行代码。

```
var findNodes = function (callback) {  
  var nodes = [];  
  
  var node = complexComputation();  
  
  // 如果回调函数可用，则执行它  
  if (typeof callback === "function") {  
    callback(node);  
  }  
  
  nodes.push(node);  
  return nodes;  
};
```

关于callback的定义，我们可以事先定义好来用：

```
// 定义callback  
var hide = function (node) {  
  node.style.display = "none";  
};  
  
// 查找node，然后隐藏所有的node  
var hiddenNodes = findNodes(hide);
```

也可以直接在调用的时候使用匿名定义，如下：

```
// 使用匿名函数定义callback
var blockNodes = findNodes(function (node) {
  node.style.display = 'block';
});
```

我们平时用的最多的，估计就数jQuery的ajax方法的调用了，通过在done/faild上定义callback，以便在ajax调用成功或者失败的时候做进一步处理，代码如下(本代码基于jquery1.8版)：

```
var menuId = $("ul.nav").first().attr("id");
var request = $.ajax({
  url: "script.php",
  type: "POST",
  data: {id : menuId},
  dataType: "html"
});

//调用成功时的回调处理
request.done(function(msg) {
  $("#log").html( msg );
});

//调用失败时的回调处理
request.fail(function(jqXHR, textStatus) {
  alert( "Request failed: " + textStatus );
});
```

配置对象

如果一个函数（或方法）的参数只有一个参数，并且参数为对象字面量，我们则称这种模式为配置对象模式。例如，如下代码：

```
var conf = {
  username:"shichuan",
  first:"Chuan",
  last:"Shi"
};
addPerson(conf);
```

则在addPerson内部，就可以随意使用conf的值了，一般用于初始化工作，例如jquery里的ajaxSetup也就是这种方式来实现的：

```
// 事先设置好初始值
```

```
$.ajaxSetup({
    url: "/xmlhttp/",
    global: false,
    type: "POST"

});

// 然后再调用
$.ajax({ data: myData });
```

另外，很多jquery的插件也有这种形式的传参，只不过也可以不传，不传的时候则就使用默认值了。

返回函数

返回函数，则是指在一个函数的返回值为另外一个函数，或者根据特定的条件灵活创建的新函数，示例代码如下：

```
var setup = function () {
    console.log(1);
    return function () {
        console.log(2);
    };
};

// 调用setup 函数
var my = setup(); // 输出 1
my(); // 输出 2
// 或者直接调用也可
setup()();
```

或者你可以利用闭包的特性，在setup函数里记录一个私有的计数器数字，通过每次调用来增加计数器，代码如下：

```
var setup = function () {
    var count = 0;
    return function () {
        return ++count;
    };
};

// 用法
var next = setup();
next(); // 返回 1
next(); // 返回 2
next(); // 返回 3
```

偏应用

这里的偏应用，其实是将参数的传入工作分开进行，在有的时候一系列的操作可能会有某一个或几个参数始终完全一样，那么我们就可以先定义一个偏函数，然后再去执行这个函数（执行时传入剩余的不同参数）。

举个例子，代码如下：

```
var partialAny = (function (aps) {

    // 该函数是你们自执行函数表达式的结果，并且赋值给了partialAny变量
    function func(fn) {
        var argsOrig = aps.call(arguments, 1);
        return function () {
            var args = [],
                argsPartial = aps.call(arguments),
                i = 0;

            // 变量所有的原始参数集， // 如果参数是partialAny._ 占位符，则使用下一个函数参数对应的值 // 否则使用原始参数里的值
            for (; i < argsOrig.length; i++) {
                args[i] = argsOrig[i] === func._
                    ? argsPartial.shift()
                    : argsOrig[i];
            }

            // 如果有任何多余的参数，则添加到尾部
            return fn.apply(this, args.concat(argsPartial));
        };
    }

    // 用于占位符设置
    func._ = {};

    return func;
})(Array.prototype.slice);
```

使用方式如下：

```
// 定义处理函数
function hex(r, g, b) {
    return '#' + r + g + b;
}

//定义偏函数，将hex的第一个参数r作为不变的参数值ff
var redMax = partialAny(hex, 'ff', partialAny._, partialAny._);

// 新函数redMax的调用方式如下，只需要传入2个参数了：
console.log(redMax('11', '22')); // "#ff1122"
```


如果觉得partialAny._太长，可以用__代替哦。

```
var __ = partialAny._;

var greenMax = partialAny(hex, __, 'ff');
console.log(greenMax('33', '44'));

var blueMax = partialAny(hex, __, __, 'ff');
console.log(blueMax('55', '66'));

var magentaMax = partialAny(hex, 'ff', __, 'ff');
console.log(magentaMax('77'));
```

这样使用，就简洁多了吧。

Currying

Currying是函数式编程的一个特性，将多个参数的处理转化成单个参数的处理，类似链式调用。

举一个简单的add函数的例子：

```
function add(x, y) {
  var oldx = x, oldy = y;
  if (typeof oldy === "undefined") { // partial
    return function (newy) {
      return oldx + newy;
    }
  }
  return x + y;
}
```

这样调用方式就可以有多种了，比如：

```
// 测试
typeof add(5); // "function"
add(3)(4); // 7

// 也可以这样调用
var add2000 = add(2000);
add2000(10); // 2010
```

接下来，我们来定义一个比较通用的currying函数：

```
// 第一个参数为要应用的function，第二个参数是需要传入的最少参数个数
function curry(func, minArgs) {
```

```

    if (minArgs == undefined) {
        minArgs = 1;
    }

    function funcWithArgsFrozen(frozenargs) {
        return function () {
            // 优化处理, 如果调用时没有参数, 返回该函数本身
            var args = Array.prototype.slice.call(arguments);
            var newArgs = frozenargs.concat(args);
            if (newArgs.length >= minArgs) {
                return func.apply(this, newArgs);
            } else {
                return funcWithArgsFrozen(newArgs);
            }
        };
    }

    return funcWithArgsFrozen([]);
}

```

这样, 我们就可以随意定义我们的业务行为了, 比如定义加法:

```

var plus = curry(function () {
    var result = 0;
    for (var i = 0; i < arguments.length; ++i) {
        result += arguments[i];
    }
    return result;
}, 2);

```

使用方式, 真实多种多样哇。

```

plus(3, 2) // 正常调用
plus(3) // 偏应用, 返回一个函数 (返回值为3+参数值)
plus(3)(2) // 完整应用 (返回5)
plus()(3)()(2) // 返回 5
plus(3, 2, 4, 5) // 可以接收多个参数
plus(3)(2, 3, 5) // 同理

```

如下是减法的例子

```

var minus = curry(function (x) {
    var result = x;
    for (var i = 1; i < arguments.length; ++i) {
        result -= arguments[i];
    }
    return result;
}, 2);

```

或者如果你想交换参数的顺序，你可以这样定义

```
var flip = curry(function (func) {  
  return curry(function (a, b) {  
    return func(b, a);  
  }, 2);  
});
```

更多资料，可以参考如下地址：

<http://www.cnblogs.com/rubylouvre/archive/2009/11/09/1598761.html>

http://www.cnblogs.com/sanshi/archive/2009/02/17/javascript_currying.html

总结

JavaScript里的Function有很多特殊的功效，可以利用闭包以及arguments参数特性实现很多不同的技巧，下一篇我们将继续介绍利用Function进行初始化的技巧。

参考地址：<http://shichuan.github.com/javascript-patterns/#function-patterns>

(50) Function模式 (下篇)

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](#)》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 = {};
```

总结

就不用总结了吧，大家仔细看代码就行咯

(结局篇)

介绍

最近几个月忙得实在是不可开交，终于把《深入理解JavaScript系列》的最后两篇“补全”了，所谓的全是不准确的，因为很多内容都没有写呢，比如高性能、Ajax安全、DOM详解、JavaScript架构等等。但因为经历所限，加上大叔希望接下来写点其它东西，所以此篇文字就暂且当前完结篇的总结吧，以后有时间的话，可以继续加上一些未涉及的专题内容。

网络文章来源

本系列文章参考了大量的互联网网站，在此向各位网站拥有者、博主、提到的以及未提到的作者们说一声：多谢感谢了。

本系列文章主要参考了如下站点：

五大原则：<http://freshbrewedcode.com/derekgreer>

ECMAScript262系列：<http://dmitrysoshnikov.com/>

DOM系列文章：<http://net.tutsplus.com>

设计模式系列文章参考如下三个网站：

<http://www.addyosmani.com/resources/essentialjsdesignpatterns/book/>

<http://shichuan.github.com/javascript-patterns/>

<https://github.com/tcorral/Design-Patterns-in-Javascript/>

其它文章，总结自自己的收藏、心得，结合了互联网上的各位大牛的博客总结整理而成，因为参考地址太多，无法一一列出，如果忘记了各位各种的版权声明，请及时告知，以便及时处理，多谢！

参考书籍

这里列出的书籍是大叔曾经读过的，也是在整理博文的时候经常参考的书籍，推荐给大家阅读。

初级读物：

《JavaScript高级程序设计》：一本非常完整的经典入门书籍，被誉为JavaScript圣经之一，

详解的非常详细，最新版第三版已经发布了，建议购买。

中级读物：

《JavaScript权威指南》：另外一本JavaScript圣经，讲解的也非常详细，属于中级读物，建议购买。

《JavaScript.The.Good.Parts》：Yahoo大牛，JavaScript精神领袖Douglas Crockford的大作，虽然才100多页，但是字字珠玑啊！强烈建议阅读。

《高性能JavaScript》：《JavaScript高级程序设计》作者Nicholas C. Zakas的又一大作。

《Eloquent JavaScript》：这本书才200多页，非常短小，但是改变了我写作的习惯，本书通过几个非常经典的例子（艾米丽姨妈的猫、悲惨的隐士、模拟生态圈、推箱子游戏等等）来介绍JavaScript方方面面的知识和应用方法，非常值得一读，同时这本书的中文版也是大叔翻译的，点击屏幕右上角可以订购，希望大家多多支持。

高级读物：

《JavaScript Patterns》：书中介绍到了各种经典的模式，如构造函数、单例、工厂等等，值得学习。

《Pro.JavaScript.Design.Patterns》：Apress出版社讲解JavaScript设计模式的书，非常不错。

《Developing JavaScript Web Applications》：构建富应用的好书，针对MVC模式有较为深入的讲解，同时也对一些流程的库进行了讲解。

《Developing Large Web Applications》：将这本书归结在这里，貌似有点不妥，因为这里不仅有JavaScript方面的介绍，还有CSS、HTML方面的介绍，但是介绍的内容却都非常不错，真正考虑到了一个大型的Web程序下，如何进行JavaScript架构设计，值得一读。

其它参考书籍：

《大话设计模式》：博文里关于设计模式的文章，有些总结性的文字来自于此。

《设计模式——可复用面向对象软件的基础》：博文里关于设计模式的文章，有些介绍性和总结性的文章来自于此。

总结

在写此系列文章期间，大叔也学到了很多很多内容。同时为了不误人子弟，大叔参考了很多很多文章，同时也阅读了那么多书籍，但博客里的文章，可能依然有很多错误，希望各位如果发现错误的话，请及时告知，以便及时修正而不再继续误导其它人。

同时，大家在阅读过程中，有任何问题都可以在相应的文章里留言，大叔将在不耽误工作的

情况下尽力回复。