编程世界里只存在两种基本元素，一个是数据，一个是代码。编程世界就是在数据和代码千丝万缕的纠缠中呈现出无限的生机和活力。

数据天生就是文静的，总想保持自己固有的本色；而代码却天生活泼，总想改变这个世界。

你看，数据代码间的关系与物质能量间的关系有着惊人的相似。数据也是有惯性的，如果没有代码来施加外力，她总保持自己原来的状态。而代码就象能量，他存在的唯一目的，就是要努力改变数据原来的状态。在代码改变数据的同时，也会因为数据的抗拒而反过来影响或改变代码原有的趋势。甚至在某些情况下，数据可以转变为代码，而代码却又有可能被转变为数据，或许还存在一个类似 E=MC2 形式的数码转换方程呢。然而，就是在数据和代码间这种即矛盾又统一的运转中，总能体现出计算机世界的规律，这些规律正是我们编写的程序逻辑。

不过，由于不同程序员有着不同的世界观，这些数据和代码看起来也就不尽相同。于是，不同世界观的程序员们运用各自的方法论，推动着编程世界的进化和发展。

众所周知，当今最流行的编程思想莫过于面向对象编程的思想。为什么面向对象的思想能迅速风靡编程世界呢？因为面向对象的思想首次把数据和代码结合成统一体，并以一个简单的对象概念呈现给编程者。这一下子就将原来那些杂乱的算法与子程序，以及纠缠不清的复杂数据结构，划分成清晰而有序的对象结构，从而理清了数据与代码在我们心中那团乱麻般的结。我们又可以有一个更清晰的思维，在另一个思想高度上去探索更加浩瀚的编程世界了。

在五祖弘忍讲授完《对象真经》之后的一天，他对众弟子们说:"经已讲完，想必尔等应该有所感悟，请各自写个偈子来看"。大弟子神秀是被大家公认为悟性最高的师兄，他的偈子写道:"身是对象树，心如类般明。朝朝勤拂拭，莫让惹尘埃!"。此偈一出，立即引起师兄弟们的轰动，大家都说写得太好了。只有火头僧慧能看后，轻轻地叹了口气，又随手在墙上写道:"对象本无根，类型亦无形。本来无一物，何处惹尘埃？"。然后摇了摇头，扬长而去。大家看了慧能的偈子都说:"写的什么乱七八糟的啊，看不懂"。师父弘忍看了神秀的诗偈也点头称赞，再看慧能的诗偈之后默然摇头。就在当天夜里，弘忍却悄悄把慧能叫到自己的禅房，将珍藏多年的软件真经传授于他，然后让他趁着月色连夜逃走...

后来，慧能果然不负师父厚望，在南方开创了禅宗另一个广阔的天空。而慧能当年带走的软件真经中就有一本是《JavaScript 真经》!

回归简单

要理解 JavaScript，你得首先放下对象和类的概念，回到数据和代码的本原。前面说过，编程世界只有数据和代码两种基本元素，而这两种元素又有着纠缠不清的关系。JavaScript 就是把数据和代码都简化到最原始的程度。

JavaScript 中的数据很简洁的。简单数据只有 undefined, null, boolean, number 和 string 这五种，而复杂数据只有一种，即 object。这就好比中国古典的朴素唯物思想，把世界最基本的元素归为金木水火土，其他复杂的物质都是由这五种基本元素组成。

JavaScript 中的代码只体现为一种形式，就是 function。

注意：以上单词都是小写的，不要和 Number, String, Object, Function 等 JavaScript 内置函数混淆了。要知道，JavaScript 语言是区分大小写的呀!

任何一个 JavaScript 的标识、常量、变量和参数都只是 unfined, null, bool, number, string, object 和 function 类型中的一种，也就 typeof 返回值表明的类型。除此之外没有其他类型了。

先说说简单数据类型吧。

undefined: 代表一切未知的事物，啥都没有，无法想象，代码也就更无法去处理了。
注意：typeof(undefined) 返回也是 undefined。
可以将 undefined 赋值给任何变量或属性，但并不意味了清除了该变量，反而会因此多了一个属性。

null: 有那么一个概念，但没有东西。无中似有，有中还无。虽难以想象，但已经可以用代码来处理了。
注意：typeof(null)返回 object，但 null 并非 object，具有 null 值的变量也并非 object。

boolean: 是就是，非就非，没有疑义。对就对，错就错，绝对明确。既能被代码处理，也可以控制代码的流程。

number: 线性的事物，大小和次序分明，多而不乱。便于代码进行批量处理，也控制代码的迭代和循环等。
注意：typeof(NaN)和 typeof(Infinity)都返回 number 。
NaN 参与任何数值计算的结构都是 NaN，而且 NaN != NaN 。
Infinity / Infinity = NaN 。

string: 面向人类的理性事物，而不是机器信号。人机信息沟通，代码据此理解人的意图等等，都靠它了。

简单类型都不是对象，JavaScript 没有将对象化的能力赋予这些简单类型。直接被赋予简单类型常量值的标识符、变量和参数都不是一个对象。

所谓"对象化”，就是可以将数据和代码组织成复杂结构的能力。JavaScript 中只有 obj

ect 类型和 function 类型提供了对象化的能力。

没有类

object 就是对象的类型。在 JavaScript 中不管多么复杂的数据和代码，都可以组织成 object 形式的对象。

但 JavaScript 却没有“类”的概念！

对于许多面向对象的程序员来说，这恐怕是 JavaScript 中最难以理解的地方。是啊，几乎任何讲面向对象的书中，第一个要讲的就是“类”的概念，这可是面向对象的支柱。这突然没有了“类”，我们就象一下子没了精神支柱，感到六神无主。看来，要放下对象和类，达到“对象本无根，类型亦无形”的境界确实是件不容易的事情啊。

这样，我们先来看一段 JavaScript 程序：

```
var life = {};
for(life.age = 1; life.age <= 3; life.age++)
{
   switch(life.age)
   {
      case 1: life.body = "卵细胞";
            life.say = function(){alert(this.age+this.body)};
            break;
      case 2: life.tail = "尾巴";
            life.gill = "腮";
            life.body = "蝌蚪";
            life.say = function(){alert(this.age+this.body+"-"+this.tail+","+this.
gill)};
            break;
      case 3: delete life.tail;
            delete life.gill;
            life.legs = "四条腿";
            life.lung = "肺";
            life.body = "青蛙";
```

```
            life.say = function(){alert(this.age+this.body+"-"+this.legs+","+thi
s.lung)};
            break;
    };
    life.say();
};
```

这段 JavaScript 程序一开始产生了一个生命对象 life，life 诞生时只是一个光溜溜的对象，没有任何属性和方法。在第一次生命过程中，它有了一个身体属性 body，并有了一个 say 方法，看起来是一个“卵细胞”。在第二次生命过程中，它又长出了“尾巴”和“腮”，有了 tail 和 gill 属性，显然它是一个“蝌蚪”。在第三次生命过程中，它的 tail 和 gill 属性消失了，但又长出了“四条腿”和“肺”，有了 legs 和 lung 属性，从而最终变成了“青蛙”。如果，你的想像力丰富的话，或许还能让它变成英俊的“王子”，娶个美丽的“公主”什么的。不过，在看完这段程序之后，请你思考一个问题：

我们一定需要类吗？

还记得儿时那个“小蝌蚪找妈妈”的童话吗？也许就在昨天晚，你的孩子刚好是在这个美丽的童话中进入梦乡的吧。可爱的小蝌蚪也就是在其自身类型不断演化过程中，逐渐变成了和妈妈一样的“类”，从而找到了自己的妈妈。这个童话故事中蕴含的编程哲理就是：对象的“类”是从无到有，又不断演化，最终又消失于无形之中的...

“类”，的确可以帮助我们理解复杂的现实世界，这纷乱的现实世界也的确需要进行分类。但如果我们的思想被“类”束缚住了，“类”也就变成了“累”。想象一下，如果一个生命对象开始的时就被规定了固定的“类”，那么它还能演化吗？蝌蚪还能变成青蛙吗？还可以给孩子们讲小蝌蚪找妈妈的故事吗？

所以，JavaScript 中没有“类”，类已化于无形，与对象融为一体。正是由于放下了“类”这个概念，JavaScript 的对象才有了其他编程语言所没有的活力。

如果，此时你的内心深处开始有所感悟，那么你已经逐渐开始理解 JavaScript 的禅机了。

函数的魔力

接下来，我们再讨论一下 JavaScript 函数的魔力吧。

JavaScript 的代码就只有 function 一种形式，function 就是函数的类型。也许其他编程语言还有 procedure 或 method 等代码概念，但在 JavaScript 里只有 function 一种形式。当我们写下一个函数的时候，只不过是建立了一个 function 类型的实体而已。请看下面的程序：

```
function myfunc()
{
   alert("hello");
};

alert(typeof(myfunc));
```

这个代码运行之后可以看到typeof(myfunc)返回的是function。以上的函数写法我们称之为"定义式"的，如果我们将其改写成下面的"变量式"的，就更容易理解了：

```
var myfunc = function ()
   {
      alert("hello");
   };

alert(typeof(myfunc));
```

这里明确定义了一个变量 myfunc，它的初始值被赋予了一个 function 的实体。因此，typeof(myfunc)返回的也是 function。其实，这两种函数的写法是等价的，除了一点细微差别，其内部实现完全相同。也就是说，我们写的这些 JavaScript 函数只是一个命了名的变量而已，其变量类型即为 function，变量的值就是我们编写的函数代码体。

聪明的你或许立即会进一步的追问：既然函数只是变量，那么变量就可以被随意赋值并用到任意地方啰？

我们来看看下面的代码：

```
var myfunc = function ()
   {
      alert("hello");
   };
myfunc(); //第一次调用 myfunc，输出 hello


myfunc = function ()
   {
      alert("yeah");
   };
myfunc(); //第二次调用 myfunc，将输出 yeah
```

这个程序运行的结果告诉我们：答案是肯定的！在第一次调用函数之后，函数变量又被赋予了新的函数代码体，使得第二次调用该函数时，出现了不同的输出。

好了，我们又来把上面的代码改成第一种定义式的函数形式：

```
function myfunc ()
{
   alert("hello");
};
myfunc(); //这里调用 myfunc，输出 yeah 而不是 hello


function myfunc ()
{
   alert("yeah");
};
myfunc(); //这里调用 myfunc，当然输出 yeah
```

按理说，两个签名完全相同的函数，在其他编程语言中应该是非法的。但在 JavaScript 中，这没错。不过，程序运行之后却发现一个奇怪的现象：两次调用都只是最后那个函数里输出的值！显然第一个函数没有起到任何作用。这又是为什么呢？

原来，JavaScript 执行引擎并非一行一行地分析和执行程序，而是一段一段地分析执行的。而且，在同一段程序的分析执行中，定义式的函数语句会被提取出来优先执行。函数定义执行完之后，才会按顺序执行其他语句代码。也就是说，在第一次调用 myfunc 之前，第一个函数语句定义的代码逻辑，已被第二个函数定义语句覆盖了。所以，两次都调用都是执行最后一个函数逻辑了。

如果把这个 JavaScript 代码分成两段，例如将它们写在一个 html 中，并用<script/>标签将其分成这样的两块：

```
<script>
   function myfunc ()
   {
      alert("hello");
   };
   myfunc(); //这里调用 myfunc，输出 hello
</script>

<script>
   function myfunc ()
   {
      alert("yeah");
   };
   myfunc(); //这里调用 myfunc，输出 yeah
</script>
```

这时，输出才是各自按顺序来的，也证明了 JavaScript 的确是一段段地执行的。

一段代码中的定义式函数语句会优先执行，这似乎有点象静态语言的编译概念。所以，这一特征也被有

些人称为：JavaScript 的“预编译”。

大多数情况下，我们也没有必要去纠缠这些细节问题。只要你记住一点：JavaScript 里的代码也是一种数据，同样可以被任意赋值和修改的，而它的值就是代码的逻辑。只是，与一般数据不同的是，函数是可以被调用执行的。

不过，如果 JavaScript 函数仅仅只有这点道行的话，这与 C++的函数指针，DELPHI 的方法指针，C#的委托相比，又有啥稀奇嘛！然而，JavaScript 函数的神奇之处还体现在另外两个方面：一是函数 function 类型本身也具有对象化的能力，二是函数 function 与对象 object 超然的结合能力。

奇妙的对象

先来说说函数的对象化能力。

任何一个函数都可以为其动态地添加或去除属性，这些属性可以是简单类型，可以是对象，也可以是其他函数。也就是说，函数具有对象的全部特征，你完全可以把函数当对象来用。其实，函数就是对象，只不过比一般的对象多了一个括号“()”操作符，这个操作符用来执行函数的逻辑。即，函数本身还可以被调用，一般对象却不可以被调用，除此之外完全相同。请看下面的代码：

```
function Sing()
{
   with(arguments.callee)
     alert(author + "：" + poem);
};
Sing.author = "李白";
Sing.poem = "汉家秦地月，流影照明妃。一上玉关道，天涯去不归…";
Sing();
Sing.author = "李战";
Sing.poem = "日出汉家天，月落阴山前。女儿琵琶怨，已唱三千年…";
Sing();
```

在这段代码中，Sing 函数被定义后，又给 Sing 函数动态地增加了 author 和 poem 属性。将 author 和 poem 属性设为不同的作者和诗句，在调用 Sing()时就能显示出不同的结果。这个示例用一种诗情画意的方式，让我们理解了 JavaScript 函数就是对象的本质，也感受到了 JavaScript 语言的优美。

好了，以上的讲述，我们应该算理解了 function 类型的东西都是和 object 类型一样的东西，这种东西被我们称为“对象”。我们的确可以这样去看待这些“对象”，因为它们既有“属性”也有“方法”嘛。但下面的代码又会让我们产生新的疑惑：

```
var anObject = {};  //一个对象
anObject.aProperty = "Property of object";  //对象的一个属性
anObject.aMethod = function(){alert("Method of object")}; //对象的一个方法
//主要看下面：
alert(anObject["aProperty"]);   //可以将对象当数组以属性名作为下标来访问属性
anObject["aMethod"]();          //可以将对象当数组以方法名作为下标来调用方法
for( var s in anObject)         //遍历对象的所有属性和方法进行迭代化处理
    alert(s + " is a " + typeof(anObject[s]));
```

同样对于 function 类型的对象也是一样：

```
var aFunction = function() {};  //一个函数
aFunction.aProperty = "Property of function";  //函数的一个属性
aFunction.aMethod = function(){alert("Method of function")}; //函数的一个方法
//主要看下面：
alert(aFunction["aProperty"]);   //可以将函数当数组以属性名作为下标来访问属性
aFunction["aMethod"]();          //可以将函数当数组以方法名作为下标来调用方法
for( var s in aFunction)         //遍历函数的所有属性和方法进行迭代化处理
    alert(s + " is a " + typeof(aFunction[s]));
```

是的，对象和函数可以象数组一样，用属性名或方法名作为下标来访问并处理。那么，它到底应该算是数组呢，还是算对象？

我们知道，数组应该算是线性数据结构，线性数据结构一般有一定的规律，适合进行统一的批量迭代操作等，有点像波。而对象是离散数据结构，适合描述分散的和个性化的东西，有点像粒子。因此，我们也可以这样问：JavaScript 里的对象到底是波还是粒子？

如果存在对象量子论，那么答案一定是：波粒二象性！

因此，JavaScript 里的函数和对象既有对象的特征也有数组的特征。这里的数组被称为"字典"，一种可以任意伸缩的名称值对儿的集合。其实， object 和 function 的内部实现就是一个字典结构，但这种字典结构却通过严谨而精巧的语法表现出了丰富的外观。正如量子力学在一些地方用粒子来解释和处理问题，而在另一些地方却用波来解释和处理问题。你也可以在需要的时候，自由选择用对象还是数组来解释和处理问题。只要善于把握 JavaScript 的这些奇妙特性，就可以编写出很多简洁而强大的代码来。

放下对象

我们再来看看 function 与 object 的超然结合吧。

在面向对象的编程世界里，数据与代码的有机结合就构成了对象的概念。自从有了对象，编程世界就被划分成两部分，一个是对象内的世界，一个是对象外的世界。对象天生具有自私的一面，外面的世界未经允许是不可访问对象内部的。对象也有大方的一面，它对外提供属性和方法，也为他人服务。不过，在这里我们要谈到一个有趣的问题，就是"对象的自我意识"。

什么？没听错吧？对象有自我意识？

可能对许多程序员来说，这的确是第一次听说。不过，请君看看 C++、C#和 Java 的 this，DELPHI 的 self，还有 VB 的 me，或许你会恍然大悟！当然，也可能只是说句"不过如此"而已。

然而，就在对象将世界划分为内外两部分的同时，对象的"自我"也就随之产生。"自我意识"是生命的最基本特征！正是由于对象这种强大的生命力，才使得编程世界充满无限的生机和活力。

但对象的"自我意识"在带给我们快乐的同时也带来了痛苦和烦恼。我们给对象赋予了太多欲望，总希望

它们能做更多的事情。然而，对象的自私使得它们互相争抢系统资源，对象的自负让对象变得复杂和臃肿，对象的自欺也往往带来挥之不去的错误和异常。我们为什么会有这么多的痛苦和烦恼呢？

为此，有一个人，在对象树下，整整想了九九八十一天，终于悟出了生命的痛苦来自于欲望，但究其欲望的根源是来自于自我意识。于是他放下了“自我”，在对象树下成了佛，从此他开始普度众生，传播真经。他的名字就叫释迦摩尼，而《JavaScript 真经》正是他所传经书中的一本。

JavaScript 中也有 this，但这个 this 却与 C++、C#或 Java 等语言的 this 不同。一般编程语言的 this 就是对象自己，而 JavaScript 的 this 却并不一定！this 可能是我，也可能是你，可能是他，反正是我中有你，你中有我，这就不能用原来的那个“自我”来理解 JavaScript 这个 this 的含义了。为此，我们必须首先放下原来对象的那个“自我”。

我们来看下面的代码：

```
function WhoAmI()       //定义一个函数 WhoAmI
{
    alert("I'm " + this.name + " of " + typeof(this));
};

WhoAmI();  //此时是 this 当前这段代码的全局对象，在浏览器中就是 window 对象，其 name 属性为空字符串。输出：I'm of object

var BillGates = {name: "Bill Gates"};
BillGates.WhoAmI = WhoAmI;  //将函数 WhoAmI 作为 BillGates 的方法。
BillGates.WhoAmI();         //此时的 this 是 BillGates。输出：I'm Bill Gates of object

var SteveJobs = {name: "Steve Jobs"};
SteveJobs.WhoAmI = WhoAmI;  //将函数 WhoAmI 作为 SteveJobs 的方法。
SteveJobs.WhoAmI();         //此时的 this 是 SteveJobs。输出：I'm Steve Jobs of object
```

```
    WhoAmI.call(BillGates);     //直接将 BillGates 作为 this，调用 WhoAmI。输出：I'm Bill Gates of object
    WhoAmI.call(SteveJobs);     //直接将 SteveJobs 作为 this，调用 WhoAmI。输出：I'm Steve Jobs of object

    BillGates.WhoAmI.call(SteveJobs);   //将 SteveJobs 作为 this，却调用 BillGates 的 WhoAmI 方法。输出：I'm Steve Jobs of object
    SteveJobs.WhoAmI.call(BillGates);   //将 BillGates 作为 this，却调用 SteveJobs 的 WhoAmI 方法。输出：I'm Bill Gates of object

    WhoAmI.WhoAmI = WhoAmI;     //将 WhoAmI 函数设置为自身的方法。
    WhoAmI.name = "WhoAmI";
    WhoAmI.WhoAmI();            //此时的 this 是 WhoAmI 函数自己。输出：I'm WhoAmI of function

    ({name: "nobody", WhoAmI: WhoAmI}).WhoAmI();    //临时创建一个匿名对象并设置属性后调用 WhoAmI 方法。输出：I'm nobody of object
```

从上面的代码可以看出，同一个函数可以从不同的角度来调用，this 并不一定是函数本身所属的对象。this 只是在任意对象和 function 元素结合时的一个概念，是种结合比起一般对象语言的默认结合更加灵活，显得更加超然和洒脱。

在 JavaScript 函数中，你只能把 this 看成当前要服务的“这个”对象。this 是一个特殊的内置参数，根据 this 参数，您可以访问到“这个”对象的属性和方法，但却不能给 this 参数赋值。在一般对象语言中，方法体代码中的 this 可以省略的，成员默认都首先是“自己”的。但 JavaScript 却不同，由于不存在“自我”，当访问“这个”对象时，this 不可省略！

JavaScript 提供了传递 this 参数的多种形式和手段，其中，象 BillGates.WhoAmI()和 SteveJobs.WhoAmI()这种形式，是传递 this 参数最正规的形式，此时的 this 就是函数所属的对象本身。而大多数

情况下，我们也几乎很少去采用那些借花仙佛的调用形式。但只我们要明白 JavaScript 的这个“自我”与其他编程语言的“自我”是不同的，这是一个放下了的“自我”，这就是 JavaScript 特有的世界观。

对象素描

已经说了许多了许多话题了，但有一个很基本的问题我们忘了讨论，那就是：怎样建立对象？

在前面的示例中，我们已经涉及到了对象的建立了。我们使用了一种被称为 JavaScript Object Notation(缩写 JSON)的形式，翻译为中文就是“JavaScript 对象表示法”。

JSON 为创建对象提供了非常简单的方法。例如，

创建一个没有任何属性的对象：

```
var o = {};
```

创建一个对象并设置属性及初始值：

```
var person = {name: "Angel", age: 18, married: false};
```

创建一个对象并设置属性和方法：

```
var speaker = {text: "Hello World", say: function(){alert(this.text)}};
```

创建一个更复杂的对象，嵌套其他对象和对象数组等：

```
  var company =
  {
     name: "Microsoft",
     product: "softwares",
     chairman: {name: "Bill Gates", age: 53, Married: true},
     employees: [{name: "Angel", age: 26, Married: false}, {name: "Hanson", a
ge: 32, Marred: true}],
      readme: function() {document.write(this.name + " product " + this.produc
t);}
```

```
};
```

JSON 的形式就是用大括“{}”号包括起来的项目列表，每一个项目间并用逗号“,”分隔，而项目就是用冒号“:”分隔的属性名和属性值。这是典型的字典表示形式，也再次表明了 JavaScript 里的对象就是字典结构。不管多么复杂的对象，都可以被一句 JSON 代码来创建并赋值。

其实，JSON 就是 JavaScript 对象最好的序列化形式，它比 XML 更简洁也更省空间。对象可以作为一个 JSON 形式的字符串，在网络间自由传递和交换信息。而当需要将这个 JSON 字符串变成一个 JavaScript 对象时，只需要使用 eval 函数这个强大的数码转换引擎，就立即能得到一个 JavaScript 内存对象。正是由于 JSON 的这种简单朴素的天生丽质，才使得她在 AJAX 舞台上成为璀璨夺目的明星。

JavaScript 就是这样，把面向对象那些看似复杂的东西，用及其简洁的形式表达出来。卸下对象浮华的浓妆，还对象一个眉目清晰！

构造对象

好了，接下我们来讨论一下对象的另一种创建方法。

除 JSON 外，在 JavaScript 中我们可以使用 new 操作符结合一个函数的形式来创建对象。例如：

```
function MyFunc() {};        //定义一个空函数
var anObj = new MyFunc();  //使用 new 操作符，借助 MyFun 函数，就创建了一个对象
```

JavaScript 的这种创建对象的方式可真有意思，如何去理解这种写法呢？

其实，可以把上面的代码改写成这种等价形式：

```
function MyFunc(){};
var anObj = {};     //创建一个对象
MyFunc.call(anObj); //将 anObj 对象作为 this 指针调用 MyFunc 函数
```

我们就可以这样理解，JavaScript 先用 new 操作符创建了一个对象，紧接着就将这个对象作为 this 参数调用了后面的函数。其实，JavaScript 内部就是这么做的，而且任何函数都可以被这样调用！但从“anObj = new MyFunc()”这种形式，我们又看到一个熟悉的身影，C++和 C#不就是这样创建对象的吗？原来，条条大路通灵山，殊途同归啊！

君看到此处也许会想，我们为什么不可以把这个 MyFunc 当作构造函数呢？恭喜你，答对了！JavaScript 也是这么想的！请看下面的代码：

```
1    function Person(name)  //带参数的构造函数
2    {
3        this.name = name;  //将参数值赋给给 this 对象的属性
4        this.SayHello = function() {alert("Hello, I'm " + this.name);};  //给 this 对象定义一个 SayHello 方法。
5    };
6
7    function Employee(name, salary)    //子构造函数
8    {
9        Person.call(this, name);       //将 this 传给父构造函数
10       this.salary = salary;      //设置一个 this 的 salary 属性
11       this.ShowMeTheMoney = function() {alert(this.name + " $" + this.salary);};  //添加 ShowMeTheMoney 方法。
12   };
13
14   var BillGates = new Person("Bill Gates");  //用 Person 构造函数创建 BillGates 对象
15   var SteveJobs = new Employee("Steve Jobs", 1234);  //用 Empolyee 构造函数创建 SteveJobs 对象
16
17   BillGates.SayHello();  //显示：I'm Bill Gates
18   SteveJobs.SayHello();  //显示：I'm Steve Jobs
19   SteveJobs.ShowMeTheMoney();  //显示：Steve Jobs $1234
```

```
20
21    alert(BillGates.constructor == Person);  //显示：true
22    alert(SteveJobs.constructor == Employee);  //显示：true
23
24    alert(BillGates.SayHello == SteveJobs.SayHello); //显示：false
```

这段代码表明，函数不但可以当作构造函数，而且还可以带参数，还可以为对象添加成员和方法。其中的第 9 行，Employee 构造函数又将自己接收的 this 作为参数调用 Person 构造函数，这就是相当于调用基类的构造函数。第 21、22 行还表明这样一个意思：BillGates 是由 Person 构造的，而 SteveJobs 是由 Employee 构造的。对象内置的 constructor 属性还指明了构造对象所用的具体函数！

其实，如果你愿意把函数当作“类”的话，她就是“类”，因为她本来就有“类”的那些特征。难道不是吗？她生出的儿子各个都有相同的特征，而且构造函数也与类同名嘛！

但要注意的是，用构造函数操作 this 对象创建出来的每一个对象，不但具有各自的成员数据，而且还具有各自的方法数据。换句话说，方法的代码体(体现函数逻辑的数据)在每一个对象中都存在一个副本。尽管每一个代码副本的逻辑是相同的，但对象们确实是各自保存了一份代码体。上例中的最后一句说明了这一实事，这也解释了 JavaScript 中的函数就是对象的概念。

同一类的对象各自有一份方法代码显然是一种浪费。在传统的对象语言中，方法函数并不象 JavaScript 那样是个对象概念。即使也有象函数指针、方法指针或委托那样的变化形式，但其实质也是对同一份代码的引用。一般的对象语言很难遇到这种情况。

不过，JavaScript 语言有大的灵活性。我们可以先定义一份唯一的方法函数体，并在构造 this 对象时使用这唯一的函数对象作为其方法，就能共享方法逻辑。例如：

```
function SayHello()    //先定义一份 SayHello 函数代码
{
    alert("Hello, I'm " + this.name);
};
```

```
function Person(name)  //带参数的构造函数
{
    this.name = name;  //将参数值赋给给 this 对象的属性
    this.SayHello = SayHello;  //给 this 对象 SayHello 方法赋值为前面那份 SayHello 代码。
};

var BillGates = new Person("Bill Gates");  //创建 BillGates 对象
var SteveJobs = new Person("Steve Jobs");  //创建 SteveJobs 对象

alert(BillGates.SayHello == SteveJobs.SayHello); //显示：true
```

其中，最后一行的输出结果表明两个对象确实共享了一个函数对象。虽然，这段程序达到了共享了一份方法代码的目的，但却不怎么优雅。因为，定义 SayHello 方法时反映不出其与 Person 类的关系。“优雅”这个词用来形容代码，也不知道是谁先提出来的。不过，这个词反映了程序员已经从追求代码的正确、高效、可靠和易读等基础上，向着追求代码的美观感觉和艺术境界的层次发展，程序人生又多了些浪漫色彩。

显然，JavaScript 早想到了这一问题，她的设计者们为此提供了一个有趣的 prototype 概念。

初看原型

prototype 源自法语，软件界的标准翻译为“原型”，代表事物的初始形态，也含有模型和样板的意义。JavaScript 中的 prototype 概念恰如其分地反映了这个词的内含，我们不能将其理解为 C++的 prototype 那种预先声明的概念。

JavaScript 的所有 function 类型的对象都有一个 prototype 属性。这个 prototype 属性本身又是一个 object 类型的对象，因此我们也可以给这个 prototype 对象添加任意的属性和方法。既然 prototype 是对象的“原型”，那么由该函数构造出来的对象应该都会具有这个“原型”的特性。事实上，在构造函数的 prototype 上定义的所有属性和方法，都是可以通过其构造的对象直接访问和调用的。也可以这么说，prototype 提供了一群同类对象共享属性和方法的机制。

我们先来看看下面的代码：

```
function Person(name)
{
    this.name = name;   //设置对象属性，每个对象各自一份属性数据
};

Person.prototype.SayHello = function()  //给 Person 函数的 prototype 添加 SayHello 方法。
{
    alert("Hello, I'm " + this.name);
}

var BillGates = new Person("Bill Gates");   //创建 BillGates 对象
var SteveJobs = new Person("Steve Jobs");   //创建 SteveJobs 对象

BillGates.SayHello();   //通过 BillGates 对象直接调用到 SayHello 方法
SteveJobs.SayHello();   //通过 SteveJobs 对象直接调用到 SayHello 方法

alert(BillGates.SayHello == SteveJobs.SayHello); //因为两个对象是共享 prototype 的 SayHello，所以显示：true
```

程序运行的结果表明，构造函数的 prototype 上定义的方法确实可以通过对象直接调用到，而且代码是共享的。显然，把方法设置到 prototype 的写法显得优雅多了，尽管调用形式没有变，但逻辑上却体现了方法与类的关系，相对前面的写法，更容易理解和组织代码。

那么，对于多层次类型的构造函数情况又如何呢？

我们再来看下面的代码：

```
1    function Person(name)   //基类构造函数
```

```
2    {
3        this.name = name;
4    };
5
6    Person.prototype.SayHello = function()  //给基类构造函数的 prototype 添加方法
7    {
8        alert("Hello, I'm " + this.name);
9    };
10
11   function Employee(name, salary) //子类构造函数
12   {
13       Person.call(this, name);    //调用基类构造函数
14       this.salary = salary;
15   };
16
17   Employee.prototype = new Person();  //建一个基类的对象作为子类原型的原型，
这里很有意思
18
19   Employee.prototype.ShowMeTheMoney = function()  //给子类添构造函数的 pr
ototype 添加方法
20   {
21       alert(this.name + " $" + this.salary);
22   };
23
24   var BillGates = new Person("Bill Gates");   //创建基类 Person 的 BillGates 对象
25   var SteveJobs = new Employee("Steve Jobs", 1234);   //创建子类 Employee
的 SteveJobs 对象
26
27   BillGates.SayHello();       //通过对象直接调用到 prototype 的方法
28   SteveJobs.SayHello();       //通过子类对象直接调用基类 prototype 的方法，关注!
```

```
29    SteveJobs.ShowMeTheMoney(); //通过子类对象直接调用子类 prototype 的方法
30
31    alert(BillGates.SayHello == SteveJobs.SayHello); //显示：true，表明 prototype 的方法是共享的
```

这段代码的第 17 行，构造了一个基类的对象，并将其设为子类构造函数的 prototype，这是很有意思的。这样做的目的就是为了第 28 行，通过子类对象也可以直接调用基类 prototype 的方法。为什么可以这样呢？

原来，在 JavaScript 中，prototype 不但能让对象共享自己财富，而且 prototype 还有寻根问祖的天性，从而使得先辈们的遗产可以代代相传。当从一个对象那里读取属性或调用方法时，如果该对象自身不存在这样的属性或方法，就会去自己关联的 prototype 对象那里寻找；如果 prototype 没有，又会去 prototype 自己关联的前辈 prototype 那里寻找，直到找到或追溯过程结束为止。

在 JavaScript 内部，对象的属性和方法追溯机制是通过所谓的 prototype 链来实现的。当用 new 操作符构造对象时，也会同时将构造函数的 prototype 对象指派给新创建的对象，成为该对象内置的原型对象。对象内置的原型对象应该是对外不可见的，尽管有些浏览器(如 Firefox)可以让我们访问这个内置原型对象，但并不建议这样做。内置的原型对象本身也是对象，也有自己关联的原型对象，这样就形成了所谓的原型链。

在原型链的最末端，就是 Object 构造函数 prototype 属性指向的那一个原型对象。这个原型对象是所有对象的最老祖先，这个老祖宗实现了诸如 toString 等所有对象天生就该具有的方法。其他内置构造函数，如 Function，Boolean，String，Date 和 RegExp 等的 prototype 都是从这个老祖宗传承下来的，但他们各自又定义了自身的属性和方法，从而他们的子孙就表现出各自宗族的那些特征。

这不就是“继承”吗？是的，这就是“继承”，是 JavaScript 特有的“原型继承”。

“原型继承”是慈祥而又严厉的。原形对象将自己的属性和方法无私地贡献给孩子们使用，也并不强迫孩子们必须遵从，允许一些顽皮孩子按自己的兴趣和爱好独立行事。从这点上看，原型对象是一位慈祥的母亲。然而，任何一个孩子虽然可以我行我素，但却不能动原型对象既有的财产，因为那可能会影响到其他

孩子的利益。从这一点上看，原型对象又象一位严厉的父亲。我们来看看下面的代码就可以理解这个意思了：

```
function Person(name)
{
    this.name = name;
};

Person.prototype.company = "Microsoft"; //原型的属性

Person.prototype.SayHello = function()  //原型的方法
{
    alert("Hello, I'm " + this.name + " of " + this.company);
};

var BillGates = new Person("Bill Gates");
BillGates.SayHello();   //由于继承了原型的东西，规规矩矩输出：Hello, I'm Bill Gates

var SteveJobs = new Person("Steve Jobs");
SteveJobs.company = "Apple";    //设置自己的 company 属性，掩盖了原型的 company 属性
SteveJobs.SayHello = function() //实现了自己的 SayHello 方法，掩盖了原型的 SayHello 方法
{
    alert("Hi, " + this.name + " like " + this.company + ", ha ha ha ");
};

SteveJobs.SayHello();   //都是自己覆盖的属性和方法，输出：Hi, Steve Jobs like Apple, ha ha ha

BillGates.SayHello();   //SteveJobs 的覆盖没有影响原型对象，BillGates 还是按老样子
```

```
输出
```

对象可以掩盖原型对象的那些属性和方法，一个构造函数原型对象也可以掩盖上层构造函数原型对象既有的属性和方法。这种掩盖其实只是在对象自己身上创建了新的属性和方法，只不过这些属性和方法与原型对象的那些同名而已。JavaScript 就是用这简单的掩盖机制实现了对象的“多态”性，与静态对象语言的虚函数和重载(override)概念不谋而合。

然而，比静态对象语言更神奇的是，我们可以随时给原型对象动态添加新的属性和方法，从而动态地扩展基类的功能特性。这在静态对象语言中是很难想象的。我们来看下面的代码：

```
function Person(name)
{
   this.name = name;
};

Person.prototype.SayHello = function()  //建立对象前定义的方法
{
   alert("Hello, I'm " + this.name);
};

var BillGates = new Person("Bill Gates");   //建立对象

BillGates.SayHello();

Person.prototype.Retire = function()    //建立对象后再动态扩展原型的方法
{
   alert("Poor " + this.name + ", bye bye!");
};

BillGates.Retire(); //动态扩展的方法即可被先前建立的对象立即调用
```

阿弥陀佛，原型继承竟然可以玩出有这样的法术！

原型扩展

想必君的悟性极高，可能你会这样想：如果在 JavaScript 内置的那些如 Object 和 Function 等函数的 prototype 上添加些新的方法和属性，是不是就能扩展 JavaScript 的功能呢？

那么，恭喜你，你得到了！

在 AJAX 技术迅猛发展的今天，许多成功的 AJAX 项目的 JavaScript 运行库都大量扩展了内置函数的 prototype 功能。比如微软的 ASP.NET AJAX，就给这些内置函数及其 prototype 添加了大量的新特性，从而增强了 JavaScript 的功能。

我们来看一段摘自 MicrosoftAjax.debug.js 中的代码：

```
String.prototype.trim = function String$trim() {
    if (arguments.length !== 0) throw Error.parameterCount();
    return this.replace(/^\s+|\s+$/g, '');
}
```

这段代码就是给内置 String 函数的 prototype 扩展了一个 trim 方法，于是所有的 String 类对象都有了 trim 方法了。有了这个扩展，今后要去除字符串两段的空白，就不用再分别处理了，因为任何字符串都有了这个扩展功能，只要调用即可，真的很方便。

当然，几乎很少有人去给 Object 的 prototype 添加方法，因为那会影响到所有的对象，除非在你的架构中这种方法的确是所有对象都需要的。

前两年，微软在设计 AJAX 类库的初期，用了一种被称为“闭包”(closure)的技术来模拟“类”。其大致模型如下：

```
function Person(firstName, lastName, age)
```

```
{
    //私有变量：
    var _firstName = firstName;
    var _lastName = lastName;

    //公共变量：
    this.age = age;

    //方法：
    this.getName = function()
    {
        return(firstName + " " + lastName);
    };
    this.SayHello = function()
    {
        alert("Hello, I'm " + firstName + " " + lastName);
    };
};

var BillGates = new Person("Bill", "Gates", 53);
var SteveJobs = new Person("Steve", "Jobs", 53);

BillGates.SayHello();
SteveJobs.SayHello();
alert(BillGates.getName() + " " + BillGates.age);
alert(BillGates.firstName);     //这里不能访问到私有变量
```

很显然，这种模型的类描述特别象 C#语言的描述形式，在一个构造函数里依次定义了私有成员、公共属性和可用的方法，显得非常优雅嘛。特别是"闭包"机制可以模拟对私有成员的保护机制，做得非常漂亮。

所谓的“闭包”，就是在构造函数体内定义另外的函数作为目标对象的方法函数，而这个对象的方法函数反过来引用外层外层函数体中的临时变量。这使得只要目标对象在生存期内始终能保持其方法，就能间接保持原构造函数体当时用到的临时变量值。尽管最开始的构造函数调用已经结束，临时变量的名称也都消失了，但在目标对象的方法内却始终能引用到该变量的值，而且该值只能通这种方法来访问。即使再次调用相同的构造函数，但只会生成新对象和方法，新的临时变量只是对应新的值，和上次那次调用的是各自独立的。的确很巧妙！

但是前面我们说过，给每一个对象设置一份方法是一种很大的浪费。还有，“闭包”这种间接保持变量值的机制，往往会给 JavaSript 的垃圾回收器制造难题。特别是遇到对象间复杂的循环引用时，垃圾回收的判断逻辑非常复杂。无独有偶，IE 浏览器早期版本确实存在 JavaSript 垃圾回收方面的内存泄漏问题。再加上“闭包”模型在性能测试方面的表现不佳，微软最终放弃了“闭包”模型，而改用“原型”模型。正所谓“有得必有失”嘛。

原型模型需要一个构造函数来定义对象的成员，而方法却依附在该构造函数的原型上。大致写法如下：

```
//定义构造函数
function Person(name)
{
   this.name = name;   //在构造函数中定义成员
};

//方法定义到构造函数的 prototype 上
Person.prototype.SayHello = function()
{
   alert("Hello, I'm " + this.name);
};

//子类构造函数
function Employee(name, salary)
{
   Person.call(this, name);    //调用上层构造函数
```

```
    this.salary = salary;       //扩展的成员
};

//子类构造函数首先需要用上层构造函数来建立 prototype 对象，实现继承的概念
Employee.prototype = new Person()  //只需要其 prototype 的方法，此对象的成员没有任何意义！

//子类方法也定义到构造函数之上
Employee.prototype.ShowMeTheMoney = function()
{
    alert(this.name + " $" + this.salary);
};

var BillGates = new Person("Bill Gates");
BillGates.SayHello();

var SteveJobs = new Employee("Steve Jobs", 1234);
SteveJobs.SayHello();
SteveJobs.ShowMeTheMoney();
```

原型类模型虽然不能模拟真正的私有变量，而且也要分两部分来定义类，显得不怎么“优雅”。不过，对象间的方法是共享的，不会遇到垃圾回收问题，而且性能优于“闭包”模型。正所谓“有失必有得”嘛。

在原型模型中，为了实现类继承，必须首先将子类构造函数的 prototype 设置为一个父类的对象实例。创建这个父类对象实例的目的就是为了构成原型链，以起到共享上层原型方法作用。但创建这个实例对象时，上层构造函数也会给它设置对象成员，这些对象成员对于继承来说是没有意义的。虽然，我们也没有给构造函数传递参数，但确实创建了若干没有用的成员，尽管其值是 undefined，这也是一种浪费啊。

唉！世界上没有完美的事情啊！

原型真谛

正当我们感概万分时，天空中一道红光闪过，祥云中出现了观音菩萨。只见她手持玉净瓶，轻拂翠柳枝，洒下几滴甘露，顿时让 JavaScript 又添新的灵气。

观音洒下的甘露在 JavaScript 的世界里凝结成块，成为了一种称为"语法甘露"的东西。这种语法甘露可以让我们编写的代码看起来更象对象语言。

要想知道这"语法甘露"为何物，就请君侧耳细听。

在理解这些语法甘露之前，我们需要重新再回顾一下 JavaScript 构造对象的过程。

我们已经知道，用 var anObject = new aFunction() 形式创建对象的过程实际上可以分为三步：第一步是建立一个新对象；第二步将该对象内置的原型对象设置为构造函数 prototype 引用的那个原型对象；第三步就是将该对象作为 this 参数调用构造函数，完成成员设置等初始化工作。对象建立之后，对象上的任何访问和操作都只与对象自身及其原型链上的那串对象有关，与构造函数再扯不上关系了。换句话说，构造函数只是在创建对象时起到介绍原型对象和初始化对象两个作用。

那么，我们能否自己定义一个对象来当作原型，并在这个原型上描述类，然后将这个原型设置给新创建的对象，将其当作对象的类呢？我们又能否将这个原型中的一个方法当作构造函数，去初始化新建的对象呢？例如，我们定义这样一个原型对象：

```
var Person =  //定义一个对象来作为原型类
{
    Create: function(name, age)  //这个当构造函数
    {
        this.name = name;
        this.age = age;
    },
    SayHello: function()  //定义方法
    {
        alert("Hello, I'm " + this.name);
    },
    HowOld: function()  //定义方法
    {
        alert(this.name + " is " + this.age + " years old.");
    }
```

```
};
```

这个 JSON 形式的写法多么象一个 C#的类啊！既有构造函数，又有各种方法。如果可以用某种形式来创建对象，并将对象的内置的原型设置为上面这个"类"对象，不就相当于创建该类的对象了吗？

但遗憾的是，我们几乎不能访问到对象内置的原型属性！尽管有些浏览器可以访问到对象的内置原型，但这样做的话就只能限定了用户必须使用那种浏览器。这也几乎不可行。

那么，我们可不可以通过一个函数对象来做媒介，利用该函数对象的 prototype 属性来中转这个原型，并用 new 操作符传递给新建的对象呢？

其实，象这样的代码就可以实现这一目标：

```
function anyfunc(){};          //定义一个函数躯壳
anyfunc.prototype = Person;    //将原型对象放到中转站 prototype
var BillGates = new anyfunc(); //新建对象的内置原型将是我们期望的原型对象
```

不过，这个 anyfunc 函数只是一个躯壳，在使用过这个躯壳之后它就成了多余的东西了，而且这和直接使用构造函数来创建对象也没啥不同，有点不爽。

可是，如果我们将这些代码写成一个通用函数，而那个函数躯壳也就成了函数内的函数，这个内部函数不就可以在外层函数退出作用域后自动消亡吗？而且，我们可以将原型对象作为通用函数的参数，让通用函数返回创建的对象。我们需要的就是下面这个形式：

```
function New(aClass, aParams)   //通用创建函数
{
   function new_()    //定义临时的中转函数壳
   {
      aClass.Create.apply(this, aParams);  //调用原型中定义的的构造函数，中转构造逻辑及构造参数
   };
   new_.prototype = aClass;   //准备中转原型对象
   return new new_();         //返回建立最终建立的对象
};

var Person =       //定义的类
```

```
{
    Create: function(name, age)
    {
        this.name = name;
        this.age = age;
    },
    SayHello: function()
    {
        alert("Hello, I'm " + this.name);
    },
    HowOld: function()
    {
        alert(this.name + " is " + this.age + " years old.");
    }
};

var BillGates = New(Person, ["Bill Gates", 53]); //调用通用函数创建对象，并以数组形式传递构造参数
BillGates.SayHello();
BillGates.HowOld();

alert(BillGates.constructor == Object);    //输出：true
```

这里的通用函数 New()就是一个“语法甘露”！这个语法甘露不但中转了原型对象，还中转了构造函数逻辑及构造参数。

有趣的是，每次创建完对象退出 New 函数作用域时，临时的 new_函数对象会被自动释放。由于 new_的 prototype 属性被设置为新的原型对象，其原来的原型对象和 new_之间就已解开了引用链，临时函数及其原来的原型对象都会被正确回收了。上面代码的最后一句证明，新创建的对象的 constructor 属性返回的是 Object 函数。其实新建的对象自己及其原型里没有 constructor 属性，那返回的只是最顶层原型对象的构造函数，即 Object。

有了 New 这个语法甘露，类的定义就很像 C#那些静态对象语言的形式了，这样的代码

显得多么文静而优雅啊！

当然，这个代码仅仅展示了"语法甘露"的概念。我们还需要多一些的语法甘露，才能实现用简洁而优雅的代码书写类层次及其继承关系。好了，我们再来看一个更丰富的示例吧：

```
//语法甘露：
var object =    //定义小写的 object 基本类，用于实现最基础的方法等
{
   isA: function(aType)  //一个判断类与类之间以及对象与类之间关系的基础方法
   {
      var self = this;
      while(self)
      {
         if (self == aType)
           return true;
         self = self.Type;
      };
      return false;
   }
};

function Class(aBaseClass, aClassDefine)   //创建类的函数，用于声明类及继承关系
{
   function class_()  //创建类的临时函数壳
   {
      this.Type = aBaseClass;   //我们给每一个类约定一个 Type 属性，引用其继承的类
      for(var member in aClassDefine)
         this[member] = aClassDefine[member];   //复制类的全部定义到当前创建的类
   };
```

```
    class_.prototype = aBaseClass;
    return new class_();
};

function New(aClass, aParams)  //创建对象的函数，用于任意类的对象创建
{
    function new_()    //创建对象的临时函数壳
    {
        this.Type = aClass;   //我们也给每一个对象约定一个 Type 属性，据此可以访问到对象所属的类
        if (aClass.Create)
          aClass.Create.apply(this, aParams);  //我们约定所有类的构造函数都叫 Create，这和 DELPHI 比较相似
    };
    new_.prototype = aClass;
    return new new_();
};

//语法甘露的应用效果：
var Person = Class(object,     //派生至 object 基本类
{
    Create: function(name, age)
    {
        this.name = name;
        this.age = age;
    },
    SayHello: function()
    {
        alert("Hello, I'm " + this.name + ", " + this.age + " years old.");
    }
```

```
});

var Employee = Class(Person,    //派生至 Person 类，是不是和一般对象语言很相似？
{
    Create: function(name, age, salary)
    {
        Person.Create.call(this, name, age);  //调用基类的构造函数
        this.salary = salary;
    },
    ShowMeTheMoney: function()
    {
        alert(this.name + " $" + this.salary);
    }
});

var BillGates = New(Person, ["Bill Gates", 53]);
var SteveJobs = New(Employee, ["Steve Jobs", 53, 1234]);
BillGates.SayHello();
SteveJobs.SayHello();
SteveJobs.ShowMeTheMoney();

var LittleBill = New(BillGates.Type, ["Little Bill", 6]);   //根据 BillGate 的类型创建 L
ittleBill
LittleBill.SayHello();

alert(BillGates.isA(Person));       //true
alert(BillGates.isA(Employee));     //false
alert(SteveJobs.isA(Person));       //true
alert(Person.isA(Employee));        //false
alert(Employee.isA(Person));        //true
```

“语法甘露”不用太多，只要那么一点点，就能改观整个代码的易读性和流畅性，从而让代码显得更优雅。有了这些语法甘露，JavaScript 就很像一般对象语言了，写起代码了感觉也就爽多了！

令人高兴的是，受这些甘露滋养的 JavaScript 程序效率会更高。因为其原型对象里既没有了毫无用处的那些对象级的成员，而且还不存在 constructor 属性体，少了与构造函数间的牵连，但依旧保持了方法的共享性。这让 JavaScript 在追溯原型链和搜索属性及方法时，少费许多工夫啊。

我们就把这种形式称为“甘露模型”吧！其实，这种“甘露模型”的原型用法才是符合 prototype 概念的本意，才是的 JavaScript 原型的真谛！

想必微软那些设计 AJAX 架构的工程师看到这个甘露模型时，肯定后悔没有早点把 AJAX 部门从美国搬到咱中国的观音庙来，错过了观音菩萨的点化。当然，我们也只能是在代码的示例中，把 Bill Gates 当作对象玩玩，真要让他放弃上帝转而皈依我佛肯定是不容易的，机缘未到啊！如果哪天你在微软新出的 AJAX 类库中看到这种甘露模型，那才是真正的缘分！

编程的快乐

在软件工业迅猛发展的今天，各式各样的编程语言层出不穷，新语言的诞生，旧语言的演化，似乎已经让我们眼花缭乱。为了适应面向对象编程的潮流，JavaScript 语言也在向完全面向对象的方向发展，新的 JavaScript 标准已经从语义上扩展了许多面向对象的新元素。与此相反的是，许多静态的对象语言也在向 JavaScript 的那种简洁而幽雅的方向发展。例如，新版本的 C#语言就吸收了 JSON 那样的简洁表示法，以及一些其他形式的 JavaScript 特性。

我们应该看到，随着 RIA(强互联应用)的发展和普及，AJAX 技术也将逐渐淡出江湖，JavaScript 也将最终消失或演化成其他形式的语言。但不管编程语言如何发展和演化，编程世界永远都会在“数据”与“代码”这千丝万缕的纠缠中保持着无限的生机。只要我们能看透这一点，我们就能很容易地学习和理解软件世界的各种新事物。不管是已熟悉的过程式编程，还是正在发展的函数式编程，以及未来量子纠缠态的大规模并行式编程，我们都有足够的法力来化解一切复杂的难题。

佛最后淡淡地说：只要我们放下那些表面的“类”，放下那些对象的“自我”，就能达到一种“对象本无根，类型亦无形”的境界，从而将自我融入到整个宇宙的生命轮循环中。我们将没有自我，也没有自私的欲望，你就是我，我就是你，你中有我，我中有你。这时，我们再看这生机勃勃的编程世界时，我们的内心将自然生起无限的慈爱之心，这种慈爱之心不是虚伪而是真诚的。关爱他人就是关爱自己，就是关爱这世界中的一切。那么，我们的心是永远快乐的，我们的程序是永远快乐的，我们的类是永远快乐的，我们的对象也是永远快乐的。这就是编程的极乐！

说到这里，在座的比丘都犹如醍醐灌顶，心中豁然开朗。看看左边这位早已喜不自禁，

再看看右边那位也是心花怒放。

蓦然回首时，唯见君拈花微笑…