第24章 最佳实践

本章内容
- 可维护的代码
- 保证代码性能
- 部署代码

自从2000以来，Web开发方面的种种规范、条例正在高速发展。Web开发过去曾是荒芜地带，里面东西还都凑合，而现在已经演化成了完整的研究规范，并建立了种种最佳实践。随着简单的网站成长为更加复杂的Web应用，同时Web爱好者成为了有收入的专业人士，Web开发的世界充满了各种关于最新技术和开发方法的信息。尤其是JavaScript，它从大量的研究和推断中获益。JavaScript的最佳实践分成若干类，并在开发过程的不同点上进行处理。

24.1 可维护性

在早期的网站中，JavaScript主要是用于小特效或者是表单验证。而今天的Web应用则会有成千上万行JavaScript代码，执行各种复杂的过程。这种演化让开发者必须得考虑到可维护性。除了秉承较传统理念的软件工程师外，还要雇佣JavaScript开发人员为公司创造价值，而他们并非仅仅按时交付产品，同时还要开发智力成果在之后不断地增加价值。

编写可维护的代码很重要，因为大部分开发人员都花费大量时间维护他人代码。很难从头开始开发新代码的，很多情况下是以他人的工作成果为基础的。确保自己代码的可维护性，以便其他开发人员在此基础上更好的开展工作。

注意可维护的代码的概念并不是JavaScript特有的。这里的很多概念都可以广泛应用于各种编程语言，当然也有某些特定于JavaScript的概念。

24.1.1 什么是可维护的代码

可维护的代码有一些特征。一般来说，如果说代码是可维护的，它需要遵循以下特点。
- **可理解性**——其他人可以接手代码并理解它的意图和一般途径，而无需原开发人员的完整解释。
- **直观性**——代码中的东西一看就能明白，不管其操作过程多么复杂。
- **可适应性**——代码以一种数据上的变化不要求完全重写的方法撰写。
- **可扩展性**——在代码架构上已考虑到在未来允许对核心功能进行扩展。

- **可调试性**——当有地方出错时，代码可以给予你足够的信息来尽可能直接地确定问题所在。

对于专业人士而言，能写出可维护的 JavaScript 代码是非常重要的技能。这正是周末改改网站的爱好者和真正理解自己作品的开发人员之间的区别。

24.1.2 代码约定

一种让代码变得可维护的简单途径是形成一套 JavaScript 代码的书写约定。绝大多数语言都开发出了各自的代码约定，只要在网上一搜就能找到大量相关文档。专业的组织为开发人员制定了详尽的代码约定试图让代码对任何人都可维护。杰出的开放源代码项目有着严格的代码约定要求，这让社区中的任何人都可以轻松地理解代码是如何组织的。

由于 JavaScript 的可适应性，代码约定对它也很重要。由于和大多数面向对象语言不同，JavaScript 并不强制开发人员将所有东西都定义为对象。语言可以支持各种编程风格，从传统面向对象式到声明式到函数式。只要快速浏览一下一些开源 JavaScript 库，就能发现好几种创建对象、定义方法和管理环境的途径。

以下小节将讨论代码约定的概论。对这些主题的解说非常重要，虽然可能的解说方式会有区别，这取决于个人需求。

1. 可读性

要让代码可维护，首先它必须可读。可读性与代码作为文本文件的格式化方式有关。可读性的大部分内容都是和代码的缩进相关的。当所有人都使用一样的缩进方式时，整个项目中的代码都会更加易于阅读。通常会使用若干空格而非制表符来进行缩进，这是因为制表符在不同的文本编辑器中显示效果不同。一种不错的、很常见的缩进大小为 4 个空格，当然你也可以使用其他数量。

可读性的另一方面是注释。在大多数编程语言中，对每个方法的注释都视为一个可行的实践。因为 JavaScript 可以在代码的任何地方创建函数，所以这点常常被忽略了。然而正因如此，在 JavaScript 中为每个函数编写文档就更加重要了。一般而言，有如下一些地方需要进行注释。

- **函数和方法**——每个函数或方法都应该包含一个注释，描述其目的和用于完成任务所可能使用的算法。陈述事先的假设也非常重要，如参数代表什么，函数是否有返回值（因为这不能从函数定义中推断出来）。
- **大段代码**——用于完成单个任务的多行代码应该在前面放一个描述任务的注释。
- **复杂的算法**——如果使用了一种独特的方式解决某个问题，则要在注释中解释你是如何做的。这不仅仅可以帮助其他浏览你代码的人，也能在下次你自己查阅代码的时候帮助理解。
- **Hack**——因为存在浏览器差异，JavaScript 代码一般会包含一些 hack。不要假设其他人在看代码的时候能够理解 hack 所要应付的浏览器问题。如果因为某种浏览器无法使用普通的方法，所以你需要用一些不同的方法，那么请将这些信息放在注释中。这样可以减少出现这种情况的可能性：有人偶然看到你的 hack，然后“修正”了它，最后重新引入了你本来修正了的错误。

缩进和注释可以带来更可读的代码，在未来则更容易维护。

2. 变量和函数命名

适当给变量和函数起名字对于增加代码可理解性和可维护性是非常重要的。由于很多 JavaScript 开发人员最初都只是业余爱好者，所以有一种使用无意义名字的倾向，诸如给变量起`"foo"`、`"bar"`等名字，给函数起`"doSomething"`这样的名字。专业 JavaScript 开发人员必须克服这些恶习以创建可维护的代码。命名的一般规则如下所示。

- 变量名应为名词如 car 或 person。
- 函数名应该以动词开始，如 getName()。返回布尔类型值的函数一般以 is 开头，如 isEnable()。
- 变量和函数都应使用合乎逻辑的名字，不要担心长度。长度问题可以通过后处理和压缩（本章后面会讲到）来缓解。

必须避免出现无法表示所包含的数据类型的无用变量名。有了合适的命名，代码阅读起来就像讲述故事一样，更容易理解。

3. 变量类型透明

由于在 JavaScript 中变量是松散类型的，很容易就忘记变量所应包含的数据类型。合适的命名方式可以一定程度上缓解这个问题，但放到所有的情况下看，还不够。有三种表示变量数据类型的方式。

第一种方式是初始化。当定义了一个变量后，它应该被初始化为一个值，来暗示它将来应该如何应用。例如，将来保存布尔类型值的变量应该初始化为 true 或者 false，将来保存数字的变量就应该初始化为一个数字，如以下例子所示：

```
//通过初始化指定变量类型
var found = false;          //布尔型
var count = -1;             //数字
var name = "";              //字符串
var person = null;          //对象
```

初始化为一个特定的数据类型可以很好的指明变量的类型。但缺点是它无法用于函数声明中的函数参数。

第二种方法是使用匈牙利标记法来指定变量类型。匈牙利标记法在变量名之前加上一个或多个字符来表示数据类型。这个标记法在脚本语言中很流行，曾经很长时间也是 JavaScript 所推崇的方式。JavaScript 中最传统的匈牙利标记法是用单个字符表示基本类型："o"代表对象，"s"代表字符串，"i"代表整数，"f"代表浮点数，"b"代表布尔型。如下所示：

```
//用于指定数据类型的匈牙利标记法
var bFound;       //布尔型
var iCount;       //整数
var sName;        //字符串
var oPerson;      //对象
```

JavaScript 中用匈牙利标记法的好处是函数参数一样可以使用。但它的缺点是让代码某种程度上难以阅读，阻碍了没有用它时代码的直观性和句子式的特质。因此，匈牙利标记法失去了一些开发者的宠爱。

最后一种指定变量类型的方式是使用类型注释。类型注释放在变量名右边，但是在初始化前面。这种方式是在变量旁边放一段指定类型的注释，如下所示：

```
//用于指定类型的类型注释
var found  /*:Boolean*/  = false;
var count  /*:int*/      = 10;
var name   /*:String*/   = "Nicholas";
var person /*:Object*/   = null;
```

类型注释维持了代码的整体可读性，同时注入了类型信息。类型注释的缺点是你不能用多行注释一次注释大块的代码，因为类型注释也是多行注释，两者会冲突，如下例所示所示：

```
//以下代码不能正确运行
/*
var found    /*:Boolean*/ = false;
var count    /*:int*/     = 10;
var name     /*:String*/  = "Nicholas";
var person   /*:Object*/ = null;
*/
```

这里，试图通过多行注释注释所有变量。类型注释与其相冲突，因为第一次出现的 /* （第二行）匹配了第一次出现的*/（第 3 行），这会造成一个语法错误。如果你想注释掉这些使用类型注释的代码行，最好在每一行上使用单行注释（很多编辑器可以帮你完成）。

这就是最常见的三种指定变量数据类型的方法。每种都有各自的优势和劣势，要自己在使用之前进行评估。最重要的是要确定哪种最适合你的项目并一致使用。

24.1.3 松散耦合

只要应用的某个部分过分依赖于另一部分，代码就是耦合过紧，难于维护。典型的问题如：对象直接引用另一个对象，并且当修改其中一个的同时需要修改另外一个。紧密耦合的软件难于维护并且需要经常重写。

因为 Web 应用所涉及的技术，有多种情况会使它变得耦合过紧。必须小心这些情况，并尽可能维护弱耦合的代码。

1. 解耦 HTML/JavaScript

一种最常见的耦合类型是 HTML/JavaScript 耦合。在 Web 上，HTML 和 JavaScript 各自代表了解决方案中的不同层次：HTML 是数据，JavaScript 是行为。因为它们天生就需要交互，所以有多种不同的方法将这两个技术关联起来。但是，有一些方法会将 HTML 和 JavaScript 过于紧密地耦合在一起。

直接写在 HTML 中的 JavaScript，使用包含内联代码的<script>元素或者是使用 HTML 属性来分配事件处理程序，都是过于紧密的耦合。请看以下代码。

```
<!-- 使用了 <script> 的紧密耦合的 HTML/JavaScript -->
<script type="text/javascript">
  document.write("Hello world!");
</script>

<!-- 使用事件处理程序属性值的紧密耦合的 HTML/JavaScript -->
<input type="button" value="Click Me" onclick="doSomething()" />
```

虽然这些从技术上来说都是正确的，但是实践中，它们将表示数据的 HTML 和定义行为的 JavaScript 紧密耦合在了一起。理想情况是，HTML 和 JavaScript 应该完全分离，并通过外部文件和使用 DOM 附加行为来包含 JavaScript。

当 HTML 和 JavaScript 过于紧密的耦合在一起时，出现 JavaScript 错误时就要先判断错误是出现在 HTML 部分还是在 JavaScript 文件中。它还会引入和代码是否可用的相关新问题。在这个例子中，可能在 doSomething()函数可用之前，就已经按下了按钮，引发了一个 JavaScript 错误。因为任何对按钮行为的更改要同时触及 HTML 和 JavaScript，因此影响了可维护性。而这些更改本该只在 JavaScript 中进行。

HTML 和 JavaScript 的紧密耦合也可以在相反的关系上成立：JavaScript 包含了 HTML。这通常会出现在使用 innerHTML 来插入一段 HTML 文本到页面上这种情况中，如下面的例子所示：

```
//将 HTML 紧密耦合到 JavaScript
function insertMessage(msg){
    var container = document.getElementById("container");
    container.innerHTML = "<div class=\"msg\"><p class=\"post\">" + msg + "</p>" +
        "<p><em>Latest message above.</em></p></div>";
}
```

一般来说，你应该避免在 JavaScript 中创建大量 HTML。再一次重申要保持层次的分离，这样可以很容易的确定错误来源。当使用上面这个例子的时候，有一个页面布局的问题，可能和动态创建的 HTML 没有被正确格式化有关。不过，要定位这个错误可能非常困难，因为你可能一般先看页面的源代码来查找那段烦人的 HTML，但是却没能找到，因为它是动态生成的。对数据或者布局的更改也会要求更改 JavaScript，这也表明了这两个层次过于紧密地耦合了。

HTML 呈现应该尽可能与 JavaScript 保持分离。当 JavaScript 用于插入数据时，尽量不要直接插入标记。一般可以在页面中直接包含并隐藏标记，然后等到整个页面渲染好之后，就可以用 JavaScript 显示该标记，而非生成它。另一种方法是进行 Ajax 请求并获取更多要显示的 HTML，这个方法可以让同样的渲染层（PHP、JSP、Ruby 等等）来输出标记，而不是直接嵌在 JavaScript 中。

将 HTML 和 JavaScript 解耦可以在调试过程中节省时间，更加容易确定错误的来源，也减轻维护的难度：更改行为只需要在 JavaScript 文件中进行，而更改标记则只要在渲染文件中。

2. 解耦 CSS/JavaScript

另一个 Web 层则是 CSS，它主要负责页面的显示。JavaScript 和 CSS 也是非常紧密相关的：他们都是 HTML 之上的层次，因此常常一起使用。但是，和 HTML 与 JavaScript 的情况一样，CSS 和 JavaScript 也可能会过于紧密地耦合在一起。最常见的紧密耦合的例子是使用 JavaScript 来更改某些样式，如下所示：

```
//CSS 对 JavaScript 的紧密耦合
element.style.color = "red";
element.style.backgroundColor = "blue";
```

由于 CSS 负责页面的显示，当显示出现任何问题时都应该只是查看 CSS 文件来解决。然而，当使用了 JavaScript 来更改某些样式的时候，比如颜色，就出现了第二个可能已更改和必须检查的地方。结果是 JavaScript 也在某种程度上负责了页面的显示，并与 CSS 紧密耦合了。如果未来需要更改样式表，CSS 和 JavaScript 文件可能都需要修改。这就给开发人员造成了维护上的噩梦。所以在这两个层次之间必须有清晰的划分。

现代 Web 应用常常要使用 JavaScript 来更改样式，所以虽然不可能完全将 CSS 和 JavaScript 解耦，但是还是能让耦合更松散的。这是通过动态更改样式类而非特定样式来实现的，如下例所示：

```
//CSS 对 JavaScript 的松散耦合
element.className = "edit";
```

通过只修改某个元素的 CSS 类，就可以让大部分样式信息严格保留在 CSS 中。JavaScript 可以更改样式类，但并不会直接影响到元素的样式。只要应用了正确的类，那么任何显示问题都可以直接追溯到 CSS 而非 JavaScript。

第二类紧密耦合仅会在 IE 中出现（但运行于标准模式下的 IE8 不会出现），它可以在 CSS 中通过表达式嵌入 JavaScript，如下例所示：

```
/* JavaScript 对 CSS 的紧密耦合 */
div {
    width: expression(document.body.offsetWidth - 10 + "px");
}
```

通常要避免使用表达式，因为它们不能跨浏览器兼容，还因为它们所引入的 JavaScript 和 CSS 之间的紧密耦合。如果使用了表达式，那么可能会在 CSS 中出现 JavaScript 错误。由于 CSS 表达式而追踪过 JavaScript 错误的开发人员，会告诉你在他们决定看一下 CSS 之前花了多长时间来查找错误。

再次提醒，好的层次划分是非常重要的。显示问题的唯一来源应该是 CSS，行为问题的唯一来源应该是 JavaScript。在这些层次之间保持松散耦合可以让你的整个应用更加易于维护。

3. 解耦应用逻辑 / 事件处理程序

每个 Web 应用一般都有相当多的事件处理程序，监听着无数不同的事件。然而，很少有能仔细得将应用逻辑从事件处理程序中分离的。请看以下例子：

```
function handleKeyPress(event){
    event = EventUtil.getEvent(event);
    if (event.keyCode == 13){
        var target = EventUtil.getTarget(event);
        var value = 5 * parseInt(target.value);
        if (value > 10){
            document.getElementById("error-msg").style.display = "block";
        }
    }
}
```

这个事件处理程序除了包含了应用逻辑，还进行了事件的处理。这种方式的问题有其双重性。首先，除了通过事件之外就再没有方法执行应用逻辑，这让调试变得困难。如果没有发生预想的结果怎么办？是不是表示事件处理程序没有被调用还是指应用逻辑失败？其次，如果一个后续的事件引发同样的应用逻辑，那就必须复制功能代码或者将代码抽取到一个单独的函数中。无论何种方式，都要作比实际所需更多的改动。

较好的方法是将应用逻辑和事件处理程序相分离，这样两者分别处理各自的东西。一个事件处理程序应该从事件对象中提取相关信息，并将这些信息传送到处理应用逻辑的某个方法中。例如，前面的代码可以被重写为：

```
function validateValue(value){
    value = 5 * parseInt(value);
    if (value > 10){
        document.getElementById("error-msg").style.display = "block";
    }
}

function handleKeyPress(event){
    event = EventUtil.getEvent(event);
    if (event.keyCode == 13){
        var target = EventUtil.getTarget(event);
        validateValue(target.value);
    }
}
```


改动过的代码合理将应用逻辑从事件处理程序中分离了出来。`handleKeyPress()` 函数确认是按下了 Enter 键（`event.keyCode` 为 13），取得了事件的目标并将 `value` 属性传递给 `validateValue()` 函数，这个函数包含了应用逻辑。注意 `validateValue()` 中没有任何东西会依赖于任何事件处理程序逻辑，它只是接收一个值，并根据该值进行其他处理。

从事件处理程序中分离应用逻辑有几个好处。首先，可以让你更容易更改触发特定过程的事件。如果最开始由鼠标点击事件触发过程，但现在按键也要进行同样处理，这种更改就很容易。其次，可以在

不附加到事件的情况下测试代码，使其更易创建单元测试或者是自动化应用流程。

以下是要牢记的应用和业务逻辑之间松散耦合的几条原则：

- 勿将 event 对象传给其他方法；只传来自 event 对象中所需的数据；
- 任何可以在应用层面的动作都应该可以在不执行任何事件处理程序的情况下进行；
- 任何事件处理程序都应该处理事件，然后将处理转交给应用逻辑。

牢记这几条可以在任何代码中都获得极大的可维护性的改进，并且为进一步的测试和开发制造了很多可能。

24.1.4 编程实践

书写可维护的 JavaScript 并不仅仅是关于如何格式化代码；它还关系到代码做什么的问题。在企业环境中创建的 Web 应用往往同时由大量人员一同创作。这种情况下的目标是确保每个人所使用的浏览器环境都有一致和不变的规则。因此，最好坚持以下一些编程实践。

1. 尊重对象所有权

JavaScript 的动态性质使得几乎任何东西在任何时间都可以修改。有人说在 JavaScript 没有什么神圣的东西，因为无法将某些东西标记为最终或恒定状态。这种状况在 ECMAScript 5 中通过引入防篡改对象（第 22 章讨论过）得以改变；不过，默认情况下所有对象都是可以修改的。在其他语言中，当没有实际的源代码的时候，对象和类是不可变的。JavaScript 可以在任何时候修改任意对象，这样就可以以不可预计的方式覆写默认的行为。因为这门语言没有强行的限制，所以对于开发者来说，这是很重要的，也是必要的。

也许在企业环境中最重要的编程实践就是尊重对象所有权，它的意思是你不能修改不属于你的对象。简单地说，如果你不负责创建或维护某个对象、它的对象或者它的方法，那么你就不能对它们进行修改。更具体地说：

- 不要为实例或原型添加属性；
- 不要为实例或原型添加方法；
- 不要重定义已存在的方法。

问题在于开发人员会假设浏览器环境按照某个特定方式运行，而对于多个人都用到的对象进行改动就会产生错误。如果某人期望叫做 stopEvent()的函数能取消某个事件的默认行为，但是你对其进行了更改，然后它完成了本来的任务，后来还追加了另外的事件处理程序，那肯定会出现问题了。其他开发人员会认为函数还是按照原来的方式执行，所以他们的用法会出错并有可能造成危害，因为他们并不知道有副作用。

这些规则不仅仅适用于自定义类型和对象，对于诸如 Object、String、document、window 等原生类型和对象也适用。此处潜在的问题可能更加危险，因为浏览器提供者可能会在不做宣布或者是不可预期的情况下更改这些对象。

著名的 Prototype JavaScript 库就出现过这种例子：它为 document 对象实现了 getElementsByClassName()方法，返回一个 Array 的实例并增加了一个 each()方法。John Resig 在他的博客上叙述了产生这个问题的一系列事件。他在帖子（http://ejohn.org/blog/getelementsbyclassname-pre-prototype-16/）中说，他发现当浏览器开始内部实现 getElementsByClassName()的时候就出现问题了，这个方法并不返回一个 Array 而是返回一个并不包含 each()方法的 NodeList。使用 Prototype 库的开发人员习惯于写这样的代码：

```
document.getElementsByClassName("selected").each(Element.hide);
```

虽然在没有原生实现 getElementsByClassName()的浏览器中可以正常运行，但对于支持的了浏览器就会产生错误，因为返回的值不同。你不能预测浏览器提供者在未来会怎样更改原生对象，所以不管用任何方式修改他们，都可能会导致将来你的实现和他们的实现之间的冲突。

所以，最佳的方法便是永远不修改不是由你所有的对象。所谓拥有对象，就是说这个对象是你创建的，比如你自己创建的自定义类型或对象字面量。而 Array、document 这些显然不是你的，它们在你的代码执行前就存在了。你依然可以通过以下方式为对象创建新的功能：

- 创建包含所需功能的新对象，并用它与相关对象进行交互；
- 创建自定义类型，继承需要进行修改的类型。然后可以为自定义类型添加额外功能。

现在很多 JavaScript 库都赞同并遵守这条开发原理，这样即使浏览器频繁更改，库本身也能继续成长和适应。

2. 避免全局量

与尊重对象所有权密切相关的是尽可能避免全局变量和函数。这也关系到创建一个脚本执行的一致的和可维护的环境。最多创建一个全局变量，让其他对象和函数存在其中。请看以下例子：

```
//两个全局量——避免！！
var name = "Nicholas";
function sayName(){
    alert(name);
}
```

这段代码包含了两个全局量：变量 name 和函数 sayName()。其实可以创建一个包含两者的对象，如下例所示：

```
//一个全局量——推荐
var MyApplication = {
    name: "Nicholas",
    sayName: function(){
        alert(this.name);
    }
};
```

这段重写的代码引入了一个单一的全局对象 MyApplication，name 和 sayName()都附加到其上。这样做消除了一些存在于前一段代码中的一些问题。首先，变量 name 覆盖了 window.name 属性，可能会与其他功能产生冲突；其次，它有助消除功能作用域之间的混淆。调用 MyApplication.sayName()在逻辑上暗示了代码的任何问题都可以通过检查定义 MyApplication 的代码来确定。

单一的全局量的延伸便是命名空间的概念，由 YUI（Yahoo! User Interface）库普及。命名空间包括创建一个用于放置功能的对象。在 YUI 的 2.x 版本中，有若干用于追加功能的命名空间。比如：

- YAHOO.util.Dom——处理 DOM 的方法；
- YAHOO.util.Event——与事件交互的方法；
- YAHOO.lang——用于底层语言特性的方法。

对于 YUI，单一的全局对象 YAHOO 作为一个容器，其中定义了其他对象。用这种方式将功能组合在一起的对象，叫做**命名空间**。整个 YUI 库便是构建在这个概念上的，让它能够在同一个页面上与其他的 JavaScript 库共存。

命名空间很重要的一部分是确定每个人都同意使用的全局对象的名字，并且尽可能唯一，让其他人

不太可能也使用这个名字。在大多数情况下，可以是开发代码的公司的名字，例如 YAHOO 或者 Wrox。你可以如下例所示开始创建命名空间来组合功能。

```
//创建全局对象
var Wrox = {};

//为 Professional JavaScript 创建命名空间
Wrox.ProJS = {};

//将书中用到的对象附加上去
Wrox.ProJS.EventUtil = { ... };
Wrox.ProJS.CookieUtil = { ... };
```

在这个例子中，Wrox 是全局量，其他命名空间在此之上创建。如果本书所有代码都放在 Wrox.ProJS 命名空间，那么其他作者也应把自己的代码添加到 Wrox 对象中。只要所有人都遵循这个规则，那么就不用担心其他人也创建叫做 EventUtil 或者 CookieUtil 的对象，因为它会存在于不同的命名空间中。请看以下例子：

```
//为 Professional Ajax 创建命名空间
Wrox.ProAjax = {};

//附加该书中所使用的其他对象
Wrox.ProAjax.EventUtil = { ... };
Wrox.ProAjax.CookieUtil = { ... };

//ProJS 还可以继续分别访问
Wrox.ProJS.EventUtil.addHandler( ... );

//以及 ProAjax
Wrox.ProAjax.EventUtil.addHandler( ... );
```

虽然命名空间会需要多写一些代码，但是对于可维护的目的而言是值得的。命名空间有助于确保代码可以在同一个页面上与其他代码以无害的方式一起工作。

3.避免与 null 进行比较

由于 JavaScript 不做任何自动的类型检查，所有它就成了开发人员的责任。因此，在 JavaScript 代码中其实很少进行类型检测。最常见的类型检测就是查看某个值是否为 null。但是，直接将值与 null 比较是使用过度的，并且常常由于不充分的类型检查导致错误。看以下例子：

```
function sortArray(values){
    if (values != null){            //避免!
        values.sort(comparator);
    }
}
```

该函数的目的是根据给定的比较子对一个数组进行排序。为了函数能正确执行，values 参数必需是数组，但这里的 if 语句仅仅检查该 values 是否为 null。还有其他的值可以通过 if 语句，包括字符串、数字，它们会导致函数抛出错误。

现实中，与 null 比较很少适合情况而被使用。必须按照所期望的对值进行检查，而非按照不被期望的那些。例如，在前面的范例中，values 参数应该是一个数组，那么就要检查它是不是一个数组，而不是检查它是否非 null。函数按照下面的方式修改会更加合适：

```
function sortArray(values){
    if (values instanceof Array){      //推荐
        values.sort(comparator);
    }
}
```

该函数的这个版本可以阻止所有非法值，而且完全用不着 null。

这种验证数组的技术在多框架的网页中不一定正确工作，因为每个框架都有其自己的全局对象，因此，也有自己的 Array 构造函数。如果你是从一个框架将数组传送到另一个框架，那么就要另外检查是否存在 sort() 方法。

如果看到了与 null 比较的代码，尝试使用以下技术替换：

- 如果值应为一个引用类型，使用 instanceof 操作符检查其构造函数；
- 如果值应为一个基本类型，使用 typeof 检查其类型；
- 如果是希望对象包含某个特定的方法名，则使用 typeof 操作符确保指定名字的方法存在于对象上。

代码中的 null 比较越少，就越容易确定代码的目的，并消除不必要的错误。

4. 使用常量

尽管 JavaScript 没有常量的正式概念，但它还是很有用的。这种将数据从应用逻辑分离出来的思想，可以在不冒引入错误的风险的同时，就改变数据。请看以下例子：

```
function validate(value){
    if (!value){
        alert("Invalid value!");
        location.href = "/errors/invalid.php";
    }
}
```

在这个函数中有两段数据：要显示给用户的信息以及 URL。显示在用户界面上的字符串应该以允许进行语言国际化的方式抽取出来。URL 也应被抽取出来，因为它们有随着应用成长而改变的倾向。基本上，有着可能由于这样那样原因会变化的这些数据，那么都会需要找到函数并在其中修改代码。而每次修改应用逻辑的代码，都可能会引入错误。可以通过将数据抽取出来变成单独定义的常量的方式，将应用逻辑与数据修改隔离开来。请看以下例子：

```
var Constants = {
    INVALID_VALUE_MSG: "Invalid value!",
    INVALID_VALUE_URL: "/errors/invalid.php"
};

function validate(value){
    if (!value){
        alert(Constants.INVALID_VALUE_MSG);
        location.href = Constants.INVALID_VALUE_URL;
    }
}
```

在这段重写过的代码中，消息和 URL 都被定义于 Constants 对象中，然后函数引用这些值。这些设置允许数据在无须接触使用它的函数的情况下进行变更。Constants 对象甚至可以完全在单独的文

件中进行定义，同时该文件可以由包含正确值的其他过程根据国际化设置来生成。

关键在于将数据和使用它的逻辑进行分离。要注意的值的类型如下所示。

- **重复值**——任何在多处用到的值都应抽取为一个常量。这就限制了当一个值变了而另一个没变的时候会造成的错误。这也包含了 CSS 类名。
- **用户界面字符串** ——任何用于显示给用户的字符串，都应被抽取出来以方便国际化。
- **URLs** ——在 Web 应用中，资源位置很容易变更，所以推荐用一个公共地方存放所有的 URL。
- **任意可能会更改的值** —— 每当你在用到字面量值的时候，你都要问一下自己这个值在未来是不是会变化。如果答案是“是”，那么这个值就应该被提取出来作为一个常量。

对于企业级的 JavaScript 开发而言，使用常量是非常重要的技巧，因为它能让代码更容易维护，并且在数据更改的同时保护代码。

24.2 性能

自从 JavaScript 诞生以来，用这门语言编写网页的开发人员有了极大的增长。与此同时，JavaScript 代码的执行效率也越来越受到关注。因为 JavaScript 最初是一个解释型语言，执行速度要比编译型语言慢得多。Chrome 是第一款内置优化引擎，将 JavaScript 编译成本地代码的浏览器。此后，主流浏览器纷纷效仿，陆续实现了 JavaScript 的编译执行。

即使到了编译执行 JavaScript 的新阶段，仍然会存在低效率的代码。不过，还是有一些方式可以改进代码的整体性能的。

24.2.1 注意作用域

第 4 章讨论了 JavaScript 中“作用域”的概念以及作用域链是如何运作的。随着作用域链中的作用域数量的增加，访问当前作用域以外的变量的时间也在增加。访问全局变量总是要比访问局部变量慢，因为需要遍历作用域链。只要能减少花费在作用域链上的时间，就能增加脚本的整体性能。

1.避免全局查找

可能优化脚本性能最重要的就是注意全局查找。使用全局变量和函数肯定要比局部的开销更大，因为要涉及作用域链上的查找。请看以下函数：

```
function updateUI(){
    var imgs = document.getElementsByTagName("img");
    for (var i=0, len=imgs.length; i < len; i++){
        imgs[i].title = document.title + " image " + i;
    }
    var msg = document.getElementById("msg");
    msg.innerHTML = "Update complete.";
}
```

该函数可能看上去完全正常，但是它包含了三个对于全局 document 对象的引用。如果在页面上有多个图片，那么 for 循环中的 document 引用就会被执行多次甚至上百次，每次都会要进行作用域链查找。通过创建一个指向 document 对象的局部变量，就可以通过限制一次全局查找来改进这个函数的性能：

```
function updateUI(){
    var doc = document;
    var imgs = doc.getElementsByTagName("img");
    for (var i=0, len=imgs.length; i < len; i++){
        imgs[i].title = doc.title + " image " + i;
    }

    var msg = doc.getElementById("msg");
    msg.innerHTML = "Update complete.";
}
```

这里，首先将 document 对象存在本地的 doc 变量中；然后在余下的代码中替换原来的 document。与原来的的版本相比，现在的函数只有一次全局查找，肯定更快。

将在一个函数中会用到多次的全局对象存储为局部变量总是没错的。

2. 避免 with 语句

在性能非常重要的地方必须避免使用 with 语句。和函数类似，with 语句会创建自己的作用域，因此会增加其中执行的代码的作用域链的长度。由于额外的作用域链查找，在 with 语句中执行的代码肯定会比外面执行的代码要慢。

必须使用 with 语句的情况很少，因为它主要用于消除额外的字符。在大多数情况下，可以用局部变量完成相同的事情而不引入新的作用域。下面是一个例子：

```
function updateBody(){
    with(document.body){
        alert(tagName);
        innerHTML = "Hello world!";
    }
}
```

这段代码中的 with 语句让 document.body 变得更容易使用。其实可以使用局部变量达到相同的效果，如下所示：

```
function updateBody(){
    var body = document.body
    alert(body.tagName);
    body.innerHTML = "Hello world!";
}
```

虽然代码稍微长了点，但是阅读起来比 with 语句版本更好，它确保让你知道 tagName 和 innerHTML 是属于哪个对象的。同时，这段代码通过将 document.body 存储在局部变量中省去了额外的全局查找。

24.2.2 选择正确方法

和其他语言一样，性能问题的一部分是和用于解决问题的算法或者方法有关的。老练的开发人员根据经验可以得知哪种方法可能获得更好的性能。很多应用在其他编程语言中的技术和方法也可以在 JavaScript 中使用。

1. 避免不必要的属性查找

在计算机科学中，算法的复杂度是使用 O 符号来表示的。最简单、最快捷的算法是常数值即 O(1)。之后，算法变得越来越复杂并花更长时间执行。下面的表格列出了 JavaScript 中常见的算法类型。

标记名称	描述
O(1) 常数	不管有多少值，执行的时间都是恒定的。一般表示简单值和存储在变量中的值
O(log *n*) 对数	总的执行时间和值的数量相关，但是要完成算法并不一定要获取每个值。例如：二分查找
O(*n*) 线性	总执行时间和值的数量直接相关。例如：遍历某个数组中的所有元素
O(n^2) 平方	总执行时间和值的数量有关，每个值至少要获取*n*次。例如：插入排序

常数值，即 O(1)，指代字面值和存储在变量中的值。符号 O(1)表示无论有多少个值，需要获取常量值的时间都一样。获取常量值是非常高效的过程。请看下面代码：

```
var value = 5;
var sum = 10 + value;
alert(sum);
```

该代码进行了四次常量值查找：数字 5，变量 value，数字 10 和变量 sum。这段代码的整体复杂度被认为是 O(1)。

在 JavaScript 中访问数组元素也是一个 O(1)操作，和简单的变量查找效率一样。所以以下代码和前面的例子效率一样：

```
var values = [5, 10];
var sum = values[0] + values[1];
alert(sum);
```

使用变量和数组要比访问对象上的属性更有效率，后者是一个 O(*n*)操作。对象上的任何属性查找都要比访问变量或者数组花费更长时间，因为必须在原型链中对拥有该名称的属性进行一次搜索。简而言之，属性查找越多，执行时间就越长。请看以下内容：

```
var values = { first: 5, second: 10};
var sum = values.first + values.second;
alert(sum);
```

这段代码使用两次属性查找来计算 sum 的值。进行一两次属性查找并不会导致显著的性能问题，但是进行成百上千次则肯定会减慢执行速度。

注意获取单个值的多重属性查找。例如，请看以下代码：

```
var query = window.location.href.substring(window.location.href.indexOf("?"));
```

在这段代码中，有 6 次属性查找：window.location.href.substring()有 3 次，window.location.href.indexOf()又有 3 次。只要数一数代码中的点的数量，就可以确定属性查找的次数了。这段代码由于两次用到了 window.location.href，同样的查找进行了两次，因此效率特别不好。

一旦多次用到对象属性，应该将其存储在局部变量中。第一次访问该值会是 O(*n*)，然而后续的访问都会是 O(1)，就会节省很多。例如，之前的代码可以如下重写：

```
var url = window.location.href;
var query = url.substring(url.indexOf("?"));
```

这个版本的代码只有 4 次属性查找，相对于原始版本节省了 33%。在更大的脚本中进行这种优化，倾向于获得更多改进。

一般来讲，只要能减少算法的复杂度，就要尽可能减少。尽可能多地使用局部变量将属性查找替换为值查找。进一步讲，如果即可以用数字化的数组位置进行访问，也可以使用命名属性（诸如 NodeList 对象），那么使用数字位置。

2. 优化循环

循环是编程中最常见的结构，在 JavaScript 程序中同样随处可见。优化循环是性能优化过程中很重要的一个部分，由于它们会反复运行同一段代码，从而自动地增加执行时间。在其他语言中对于循环优化有大量研究，这些技术也可以应用于 JavaScript。一个循环的基本优化步骤如下所示。

(1) **减值迭代**——大多数循环使用一个从 0 开始、增加到某个特定值的迭代器。在很多情况下，从最大值开始，在循环中不断减值的迭代器更加高效。

(2) **简化终止条件**——由于每次循环过程都会计算终止条件，所以必须保证它尽可能快。也就是说避免属性查找或其他 O(*n*)的操作。

(3) **简化循环体**——循环体是执行最多的，所以要确保其被最大限度地优化。确保没有某些可以被很容易移出循环的密集计算。

(4) **使用后测试循环**——最常用 for 循环和 while 循环都是前测试循环。而如 do-while 这种后测试循环，可以避免最初终止条件的计算，因此运行更快。

用一个例子来描述这种改动。以下是一个基本的 for 循环：

```
for (var i=0; i < values.length; i++){
    process(values[i]);
}
```

这段代码中变量 i 从 0 递增到 values 数组中的元素总数。假设值的处理顺序无关紧要，那么循环可以改为 i 减值，如下所示：

```
for (var i=values.length -1; i >= 0; i--){
    process(values[i]);
}
```

这里，变量 i 每次循环之后都会减 1。在这个过程中，将终止条件从 value.length 的 O(*n*)调用简化成了 0 的 O(1)调用。由于循环体只有一个语句，无法进一步优化。不过循环还能改成后测试循环，如下：

```
var i=values.length -1;
if (i > -1){
    do {
        process(values[i]);
    }while(--i >= 0);
}
```

此处主要的优化是将终止条件和自减操作符组合成了单个语句。这时，任何进一步的优化只能在 process()函数中进行了，因为循环部分已经优化完全了。

记住使用“后测试”循环时必须确保要处理的值至少有一个。空数组会导致多余的一次循环而“前测试”循环则可以避免。

3. 展开循环

当循环的次数是确定的，消除循环并使用多次函数调用往往更快。请看一下前面的例子。如果数组的长度总是一样的，对每个元素都调用 process()可能更优，如以下代码所示：

```
//消除循环
process(values[0]);
process(values[1]);
process(values[2]);
```

这个例子假设 values 数组里面只有 3 个元素，直接对每个元素调用 process()。这样展开循环可以消除建立循环和处理终止条件的额外开销，使代码运行得更快。

如果循环中的迭代次数不能事先确定，那可以考虑使用一种叫做 Duff 装置的技术。这个技术是以其创建者 Tom Duff 命名的，他最早在 C 语言中使用这项技术。正是 Jeff Greenberg 用 JavaScript 实现了 Duff 装置。Duff 装置的基本概念是通过计算迭代的次数是否为 8 的倍数将一个循环展开为一系列语句。请看以下代码：

```
//credit: Jeff Greenberg for JS implementation of Duff's Device
//假设 values.length > 0
var iterations = Math.ceil(values.length / 8);
var startAt = values.length % 8;
var i = 0;

do {
    switch(startAt){
        case 0: process(values[i++]);
        case 7: process(values[i++]);
        case 6: process(values[i++]);
        case 5: process(values[i++]);
        case 4: process(values[i++]);
        case 3: process(values[i++]);
        case 2: process(values[i++]);
        case 1: process(values[i++]);
    }
    startAt = 0;
} while (--iterations > 0);
```

Duff 装置的实现是通过将 values 数组中元素个数除以 8 来计算出循环需要进行多少次迭代的。然后使用取整的上限函数确保结果是整数。如果完全根据除 8 来进行迭代，可能会有一些不能被处理到的元素，这个数量保存在 startAt 变量中。首次执行该循环时，会检查 StartAt 变量看有需要多少额外调用。例如，如果数组中有 10 个值，startAt 则等于 2，那么最开始的时候 process()则只会被调用 2 次。在接下来的循环中，startAt 被重置为 0，这样之后的每次循环都会调用 8 次 process()。展开循环可以提升大数据集的处理速度。

由 Andrew B. King 所著的 *Speed Up Your Site*（New Riders，2003）提出了一个更快的 Duff 装置技术，将 do-while 循环分成 2 个单独的循环。以下是例子：

```
//credit: Speed Up Your Site (New Riders, 2003)
var iterations = Math.floor(values.length / 8);
var leftover = values.length % 8;
var i = 0;

if (leftover > 0){
    do {
        process(values[i++]);
    } while (--leftover > 0);
}
do {
    process(values[i++]);
```

```
        process(values[i++]);
        process(values[i++]);
        process(values[i++]);
        process(values[i++]);
        process(values[i++]);
        process(values[i++]);
        process(values[i++]);
    } while (--iterations > 0);
```

在这个实现中，剩余的计算部分不会在实际循环中处理，而是在一个初始化循环中进行除以 8 的操作。当处理掉了额外的元素，继续执行每次调用 8 次 process() 的主循环。这个方法几乎比原始的 Duff 装置实现快上 40%。

针对大数据集使用展开循环可以节省很多时间，但对于小数据集，额外的开销则可能得不偿失。它是要花更多的代码来完成同样的任务，如果处理的不是大数据集，一般来说并不值得。

4. 避免双重解释

当 JavaScript 代码想解析 JavaScript 的时候就会存在双重解释惩罚。当使用 eval() 函数或者是 Function 构造函数以及使用 setTimeout() 传一个字符串参数时都会发生这种情况。下面有一些例子：

```
//某些代码求值——避免!!
eval("alert('Hello world!')");

//创建新函数——避免!!
var sayHi = new Function("alert('Hello world!')");

//设置超时——避免!!
setTimeout("alert('Hello world!')", 500);
```

在以上这些例子中，都要解析包含了 JavaScript 代码的字符串。这个操作是不能在初始的解析过程中完成的，因为代码是包含在字符串中的，也就是说在 JavaScript 代码运行的同时必须新启动一个解析器来解析新的代码。实例化一个新的解析器有不容忽视的开销，所以这种代码要比直接解析慢得多。

对于这几个例子都有另外的办法。只有极少的情况下 eval() 是绝对必须的，所以尽可能避免使用。在这个例子中，代码其实可以直接内嵌在原代码中。对于 Function 构造函数，完全可以直接写成一般的函数，调用 setTimeout() 可以传入函数作为第一个参数。以下是一些例子：

```
//已修正
alert('Hello world!');

//创建新函数——已修正
var sayHi = function(){
    alert('Hello world!');
};

//设置一个超时——已修正
setTimeout(function(){
    alert('Hello world!');
}, 500);
```

如果要提高代码性能，尽可能避免出现需要按照 JavaScript 解释的字符串。

5. 性能的其他注意事项

当评估脚本性能的时候，还有其他一些可以考虑的东西。下面并非主要的问题，不过如果使用得当也会有相当大的提升。

- **原生方法较快**——只要有可能，使用原生方法而不是自己用 JavaScript 重写一个。原生方法是用诸如 C/C++之类的编译型语言写出来的，所以要比 JavaScript 的快很多很多。JavaScript 中最容易被忘记的就是可以在 Math 对象中找到的复杂的数学运算；这些方法要比任何用 JavaScript 写的同样方法如正弦、余弦快的多。
- **Switch 语句较快** —— 如果有一系列复杂的 if-else 语句，可以转换成单个 switch 语句则可以得到更快的代码。还可以通过将 case 语句按照最可能的到最不可能的顺序进行组织，来进一步优化 switch 语句。
- **位运算符较快** —— 当进行数学运算的时候，位运算操作要比任何布尔运算或者算数运算快。选择性地用位运算替换算数运算可以极大提升复杂计算的性能。诸如取模，逻辑与和逻辑或都可以考虑用位运算来替换。

24.2.3 最小化语句数

JavaScript 代码中的语句数量也影响所执行的操作的速度。完成多个操作的单个语句要比完成单个操作的多个语句快。所以，就要找出可以组合在一起的语句，以减少脚本整体的执行时间。这里有几个可以参考的模式。

1. 多个变量声明

有个地方很多开发人员都容易创建很多语句，那就是多个变量的声明。很容易看到代码中由多个 var 语句来声明多个变量，如下所示：

```
//4 个语句——很浪费
var count = 5;
var color = "blue";
var values = [1,2,3];
var now = new Date();
```

在强类型语言中，不同的数据类型的变量必须在不同的语句中声明。然而，在 JavaScript 中所有的变量都可以使用单个 var 语句来声明。前面的代码可以如下重写：

```
//一个语句
var count = 5,
    color = "blue",
    values = [1,2,3],
    now = new Date();
```

此处，变量声明只用了一个 var 语句，之间由逗号隔开。在大多数情况下这种优化都非常容易做，并且要比单个变量分别声明快很多。

2. 插入迭代值

当使用迭代值（也就是在不同的位置进行增加或减少的值）的时候，尽可能合并语句。请看以下代码：

```
var name = values[i];
i++;
```

前面这 2 句语句各只有一个目的：第一个从 values 数组中获取值，然后存储在 name 中；第二个给变量 i 增加 1。这两句可以通过迭代值插入第一个语句组合成一个语句，如下所示：

```
var name = values[i++];
```

这一个语句可以完成和前面两个语句一样的事情。因为自增操作符是后缀操作符，i 的值只有在语句其他部分结束之后才会增加。一旦出现类似情况，都要尝试将迭代值插入到最后使用它的语句中去。

3. 使用数组和对象字面量

本书中，你可能看过两种创建数组和对象的方法：使用构造函数或者是使用字面量。使用构造函数总是要用到更多的语句来插入元素或者定义属性，而字面量可以将这些操作在一个语句中完成。请看以下例子：

```
//用 4 个语句创建和初始化数组——浪费
var values = new Array();
values[0] = 123;
values[1] = 456;
values[2] = 789;

//用 4 个语句创建和初始化对象——浪费
var person = new Object();
person.name = "Nicholas";
person.age = 29;
person.sayName = function(){
    alert(this.name);
};
```

这段代码中，只创建和初始化了一个数组和一个对象。各用了 4 个语句：一个调用构造函数，其他 3 个分配数据。其实可以很容易地转换成使用字面量的形式，如下所示：

```
//只用一条语句创建和初始化数组
var values = [123, 456, 789];

//只用一条语句创建和初始化对象
var person = {
    name : "Nicholas",
    age : 29,
    sayName : function(){
        alert(this.name);
    }
};
```

重写后的代码只包含两条语句，一条创建和初始化数组，另一条创建和初始化对象。之前用了八条语句的东西现在只用了两条，减少了 75%的语句量。在包含成千上万行 JavaScript 的代码库中，这些优化的价值更大。

只要有可能，尽量使用数组和对象的字面量表达方式来消除不必要的语句。

在 IE6 和更早版本中使用字面量有微小的性能惩罚。不过这些问题在 IE7 中已经解决。

24.2.4 优化 DOM 交互

在 JavaScript 各个方面中，DOM 毫无疑问是最慢的一部分。DOM 操作与交互要消耗大量时间，因为它们往往需要重新渲染整个页面或者某一部分。进一步说，看似细微的操作也可能要花很久来执行，因为 DOM 要处理非常多的信息。理解如何优化与 DOM 的交互可以极大得提高脚本完成的速度。

1. 最小化现场更新

一旦你需要访问的 DOM 部分是已经显示的页面的一部分，那么你就是在进行一个*现场更新*。之所以叫现场更新，是因为需要立即（现场）对页面对用户的显示进行更新。每一个更改，不管是插入单个字符，还是移除整个片段，都有一个性能惩罚，因为浏览器要重新计算无数尺寸以进行更新。现场更新进行得越多，代码完成执行所花的时间就越长；完成一个操作所需的现场更新越少，代码就越快。请看以下例子：

```
var list = document.getElementById("myList"),
               item,
               i;

for (i=0; i < 10; i++) {
    item = document.createElement("li");
    list.appendChild(item);
    item.appendChild(document.createTextNode("Item " + i));
}
```

这段代码为列表添加了 10 个项目。添加每个项目时，都有 2 个现场更新：一个添加<li>元素，另一个给它添加文本节点。这样添加 10 个项目，这个操作总共要完成 20 个现场更新。

要修正这个性能瓶颈，需要减少现场更新的数量。一般有 2 种方法。第一种是将列表从页面上移除，最后进行更新，最后再将列表插回到同样的位置。这个方法不是非常理想，因为在每次页面更新的时候它会不必要的闪烁。第二个方法是使用文档碎片来构建 DOM 结构，接着将其添加到 List 元素中。这个方式避免了现场更新和页面闪烁问题。请看下面内容：

```
var list = document.getElementById("myList"),
    fragment = document.createDocumentFragment(),
    item,
    i;

for (i=0; i < 10; i++) {
    item = document.createElement("li");
    fragment.appendChild(item);
    item.appendChild(document.createTextNode("Item " + i));
}

list.appendChild(fragment);
```

在这个例子中只有一次现场更新，它发生在所有项目都创建好之后。文档碎片用作一个临时的占位符，放置新创建的项目。然后使用 appendChild()将所有项目添加到列表用。记住，当给 appendChild()传入文档碎片时，只有碎片中的子节点被添加到目标，碎片本身不会被添加的。

一旦需要更新 DOM，请考虑使用文档碎片来构建 DOM 结构，然后再将其添加到现存的文档中。

2. 使用 innerHTML

有两种在页面上创建 DOM 节点的方法：使用诸如 createElement()和 appendChild()之类的 DOM 方法，以及使用 innerHTML。对于小的 DOM 更改而言，两种方法效率都差不多。然而，对于大的 DOM 更改，使用 innerHTML 要比使用标准 DOM 方法创建同样的 DOM 结构快得多。

当把 innerHTML 设置为某个值时，后台会创建一个 HTML 解析器，然后使用内部的 DOM 调用来创建 DOM 结构，而非基于 JavaScript 的 DOM 调用。由于内部方法是编译好的而非解释执行的，所以执行快得多。前面的例子还可以用 innerHTML 改写如下：

```
var list = document.getElementById("myList"),
    html = "",
    i;

for (i=0; i < 10; i++) {
    html += "<li>Item " + i + "</li>";
}

list.innerHTML = html;
```

这段代码构建了一个 HTML 字符串，然后将其指定到 list.innerHTML，便创建了需要的 DOM 结构。虽然字符串连接上总是有点性能损失，但这种方式还是要比进行多个 DOM 操作更快。

使用 innerHTML 的关键在于（和其他 DOM 操作一样）最小化调用它的次数。例如，下面的代码在这个操作中用到 innerHTML 的次数太多了：

```
var list = document.getElementById("myList"),
    i;

for (i=0; i < 10; i++) {
    list.innerHTML += "<li>Item " + i + "</li>";          //避免!!!
}
```

这段代码的问题在于每次循环都要调用 innerHTML，这是极其低效的。调用 innerHTML 实际上就是一次现场更新，所以也要如此对待。构建好一个字符串然后一次性调用 innerHTML 要比调用 innerHTML 多次快得多。

3. 使用事件代理

大多数 Web 应用在用户交互上大量用到事件处理程序。页面上的事件处理程序的数量和页面响应用户交互的速度之间有个负相关。为了减轻这种惩罚，最好使用事件代理。

事件代理，如第 13 章中所讨论的那样，用到了事件冒泡。任何可以冒泡的事件都不仅仅可以在事件目标上进行处理，目标的任何祖先节点上也能处理。使用这个知识，就可以将事件处理程序附加到更高层的地方负责多个目标的事件处理。如果可能，在文档级别附加事件处理程序，这样可以处理整个页面的事件。

4. 注意 HTMLCollection

HTMLCollection 对象的陷阱已经在本书中讨论过了，因为它们对于 Web 应用的性能而言是巨大的损害。记住，任何时候要访问 HTMLCollection，不管它是一个属性还是一个方法，都是在文档上进行一个查询，这个查询开销很昂贵。最小化访问 HTMLCollection 的次数可以极大地改进脚本的性能。

也许优化 HTMLCollection 访问最重要的地方就是循环了。前面提到过将长度计算移入 for 循环的初始化部分。现在看一下这个例子：

```
var images = document.getElementsByTagName("img"),
    i, len;

for (i=0, len=images.length; i < len; i++){
    //处理
}
```

这里的关键在于长度 length 存入了 len 变量，而不是每次都去访问 HTMLCollection 的 length 属性。当在循环中使用 HTMLCollection 的时候，下一步应该是获取要使用的项目的引用，如下所示，以便避免在循环体内多次调用 HTMLCollection。

```
var images = document.getElementsByTagName("img"),
    image,
    i, len;

for (i=0, len=images.length; i < len; i++){
    image = images[i];
    //处理
}
```

这段代码添加了 image 变量，保存了当前的图像。这之后，在循环内就没有理由再访问 images 的 HTMLCollection 了。

编写 JavaScript 的时候，一定要知道何时返回 HTMLCollection 对象，这样你就可以最小化对他们的访问。发生以下情况时会返回 HTMLCollection 对象：

- 进行了对 getElementsByTagName() 的调用；
- 获取了元素的 childNodes 属性；
- 获取了元素的 attributes 属性；
- 访问了特殊的集合，如 document.forms、document.images 等。

要了解当使用 HTMLCollection 对象时，合理使用会极大提升代码执行速度。

24.3 部署

也许所有 JavaScript 解决方案最重要的部分，便是最后部署到运营中的网站或者是 Web 应用的过程。在这之前可能你已经做了相当多的工作，为普通的使用进行架构并优化一个解决方案。现在是时候从开发环境中走出来并进入 Web 阶段了，在此将会和真正的用户交互。然而，在这之前还有一系列需要解决的问题。

24.3.1 构建过程

完备 JavaScript 代码可以用于部署的一件很重要的事情，就是给它开发某些类型的构建过程。软件开发的典型模式是写代码-编译-测试，即首先书写好代码，将其编译通过，然后运行并确保其正常工作。由于 JavaScript 并非一个编译型语言，模式变成了写代码-测试，这里你写的代码就是你要在浏览器中测试的代码。这个方法的问题在于它不是最优的，你写的代码不应该原封不动地放入浏览器中，理由如下所示。

- **知识产权问题** —— 如果把带有完整注释的代码放到线上，那别人就更容易知道你的意图，对它再利用，并且可能找到安全漏洞。
- **文件大小** —— 书写代码要保证容易阅读，才能更好地维护，但是这对于性能是不利的。浏览器并不能从额外的空白字符或者是冗长的函数名和变量名中获得什么好处。
- **代码组织** —— 组织代码要考虑到可维护性并不一定是传送给浏览器的最好方式。

基于这些原因，最好给 JavaScript 文件定义一个构建过程。

构建过程始于在源控制中定义用于存储文件的逻辑结构。最好避免使用一个文件存放所有的 JavaScript，遵循以下面向对象语言中的典型模式：将每个对象或自定义类型分别放入其单独的文件中。这样可以确保每个文件包含最少量的代码，使其在不引入错误的情况下更容易修改。另外，在使用像 CVS 或 Subversion 这类并发源控制系统的时候，这样做也减少了在合并操作中产生冲突的风险。

记住将代码分离成多个文件只是为了提高可维护性，并非为了部署。要进行部署的时候，需要将这些源代码合并为一个或几个归并文件。推荐 Web 应用中尽可能使用最少的 JavaScript 文件，是因为 HTTP 请求是 Web 中的主要性能瓶颈之一。记住通过<script>标记引用 JavaScript 文件是一个阻塞操作，当代码下载并运行的时候会停止其他所有的下载。因此，尽量从逻辑上将 JavaScript 代码分组成部署文件。

一旦组织好文件和目录结构，并确定哪些要出现在部署文件中，就可以创建构建系统了。Ant 构建工具（http://ant.apache.org）是为了自动化 Java 构建过程而诞生的，不过因为其易用性和应用广泛，而在 Web 应用开发人员中也颇流行，诸如 Julien Lecomte 的软件工程师，已经写了教程指导如何使用 Ant 进行 JavaScript 和 CSS 的构建自动化（Lecomte 的文章在 www.julienlecomte.net/blog/2007/09/16/ ）。

Ant 由于其简便的文件处理能力而非常适合 JavaScript 编译系统。例如，可以很方便地获得目录中的所有文件的列表，然后将其合并为一个文件，如下所示：

```
<project name="JavaScript Project" default="js.concatenate">

    <!-- 输出的目录.-->
    <property name="build.dir" value="./js" />

    <!-- 包含源文件的目录 -->
    <property name="src.dir" value="./dev/src" />

    <!-- 合并所有 JS 文件的目标 -->
    <!-- Credit: Julien Lecomte, http://www.julienlecomte.net/blog/2007/09/16/ -->
    <target name="js.concatenate">
        <concat destfile="${build.dir}/output.js">
            <filelist dir="${src.dir}/js" files="a.js, b.js"/>
            <fileset dir="${src.dir}/js" includes="*.js" excludes="a.js, b.js"/>
        </concat>
    </target>

</project>
```

SampleAntDir/build.xml

该 build.xml 文件定义了两个属性：输出最终文件的构建目录，以及 JavaScript 源文件所在的源目录。目标 `js.concatenate` 使用了<concat>元素来指定需要进行合并的文件的列表以及结果文件所要输出的位置。<filelist>元素用于指定 a.js 和 b.js 要首先出现在合并的文件中，<fileset>元素指定了之后要添加到目录中的其他所有文件，a.js 和 b.js 除外。结果文件最后输出到/js/output.js。

如果安装了 Ant，就可以进入 build.xml 文件所在的目录，并运行以下命令：

```
ant
```

然后构建过程就开始了，最后生成合并了的文件。如果在文件中还有其他目标，可以使用以下代码仅执行 `js.concatenate` 目标：

```
ant js.concatenate
```

可以根据需求，修改构建过程以包含其他步骤。在开发周期中引入构建这一步能让你在部署之前对 JavaScript 文件进行更多的处理。

24.3.2 验证

尽管现在出现了一些可以理解并支持 JavaScript 的 IDE，大多数开发人员还是要在浏览器中运行代

码以检查其语法。这种方法有一些问题。首先，验证过程难以自动化或者在不同系统间直接移植。其次，除了语法错误外，很多问题只有在执行代码的时候才会遇到，这给错误留下了空间；有些工具可以帮助确定 JavaScript 代码中潜在的问题，其中最著名的就是 Douglas Crockford 的 JSLint (www.jslint.com)。

JSLint 可以查找 JavaScript 代码中的语法错误以及常见的编码错误。它可以发掘的一些潜在问题如下：

- eval()的使用；
- 未声明变量的使用；
- 遗漏的分号；
- 不恰当的换行；
- 错误的逗号使用；
- 语句周围遗漏的括号；
- switch 分支语句中遗漏的 break；
- 重复声明的变量；
- with 的使用；
- 错误使用的等号（替代了双等号或三等号）；
- 无法到达的代码。

为了方便访问，它有一个在线版本，不过它也可以使用基于 Java 的 Rhino JavaScript 引擎(www.mozilla.org/rhino/) 运行于命令行模式下。要在命令行中运行 JSLint，首先要下载 Rhino，并从 www.jslint.com/下载 Rhino 版本的 JSLint。一旦安装完成，便可以使用下面的语法从命令行运行 JSLint 了：

```
java -jar rhino-1.6R7.jar jslint.js [input files]
```

如这个例子：

```
java -jar rhino-1.6R7.jar jslint.js a.js b.js c.js
```

如果给定文件中有任何语法问题或者是潜在的错误，则会输出有关错误和警告的报告。如果没有问题，代码会直接结束而不显示任何信息。

可以使用 Ant 将 JSLint 作为构建过程的一部分运行，添加如下一个目标：

```
<target name="js.verify">
    <apply executable="java" parallel="false">
        <fileset dir="${build.dir}" includes="output.js"/>
        <arg line="-jar"/>
        <arg path="${rhino.jar}"/>
        <arg path="${jslint.js}" />
        <srcfile/>
     </apply>
</target>
```

SampleAntDir/build.xml

这个目标假设 Rhino jar 文件的位置已经由叫做 rhino.jar 的属性指定了，同时 JSLint Rhino 文件的位置由叫做 jslint.js 的属性指定了。output.js 文件被传递给 JSLint 进行校验，然后显示找到的任何问题。

给开发周期添加代码验证这个环节有助于避免将来可能出现的一些错误。建议开发人员给构建过程加入某种类型的代码验证作为确定潜在问题的一个方法，防患于未然。

JavaScript 代码校验工具的列表可以在附录 D 中找到。

24.3.3 压缩

当谈及 JavaScript 文件压缩，其实在讨论两个东西：代码长度和配重（Wire weight）。代码长度指的是浏览器所需解析的字节数，配重指的是实际从服务器传送到浏览器的字节数。在 Web 开发的早期，这两个数字几乎是一样的，因为从服务器端到客户端原封不动地传递了源文件。而在今天的 Web 上，这两者很少相等，实际上也不应相等。

1. 文件压缩

因为 JavaScript 并非编译为字节码，而是按照源代码传送的，代码文件通常包含浏览器执行所不需要的额外的信息和格式。注释，额外的空白，以及长长的变量名和函数名虽然提高了可读性，但却是传送给浏览器时不必要的字节。不过，我们可以使用压缩工具减少文件的大小。

压缩器一般进行如下一些步骤：

- 删除额外的空白（包括换行）；
- 删除所有注释；
- 缩短变量名。

JavaScript 有不少压缩工具可用（附录 D 中有一个完整列表），其中最优秀的（有争议的）是 YUI 压缩器，http://yuilibrary.com/projects/yuicompressor。YUI 压缩器使用了 Rhino JavaScript 解析器将 JavaScript 代码令牌化。然后使用这个令牌流创建代码不包含空白和注释的优化版本。与一般的基于表达式的压缩器不同的地方在于，YUI 压缩可以确保不引入任何语法错误，并可以安全地缩短局部变量名。

YUI 压缩器是作为 Java 的一个 `jar` 文件发布的，名字叫 `yuicompressor-x.y.z.jar`，其中 `x.y.z` 是版本号。在写本书的时候，`2.3.5` 是最新的版本。可以使用以下命令行格式来使用 YUI 压缩器：

```
java -jar yuicompressor-x.y.z.jar [options] [input files]
```

YUI 压缩器的选项列在了下面的表格内。

选　项	描　述
`-h`	显示帮助信息
`-o outputFile`	指定输出文件的文件名。如果没有该选项，那么输出文件名是输入文件名加上-min。例如，叫做 input.js 的输入文件，那么会产生 input-min.js
`--line-break column`	指定每行多少个字符之后添加换行。默认情况下，压缩过的文件只输出为一行，可能在某些版本控制系统中会出错
`-v, --verbose`	详细模式，输出可以进行更好压缩的提示和警告
`--charset charset`	指定输入文件所使用的字符集。输出文件会使用同样的字符集
`--nomunge`	关闭局部变量替换
`--disable-optimizations`	关闭 YUI 压缩器的细节优化
`--preserve-semi`	保留本来要被删除的无用的分号

例如，以下命令行可以用来将 CookieUtil.js 压缩成一个叫做 cookie.js 的文件：

```
java -jar yuicompressor-2.3.5.jar -o cookie.js CookieUtil.js
```

YUI 压缩器也可以通过直接调用 java 可执行文件在 Ant 中使用，如下面的例子所示：

```
<!-- Credit: Julien Lecomte, http://www.julienlecomte.net/blog/2007/09/16/ -->
<target name="js.compress">
    <apply executable="java" parallel="false">
        <fileset dir="${build.dir}" includes="output.js"/>
        <arg line="-jar"/>
        <arg path="${yuicompressor.jar}"/>
        <arg line="-o ${build.dir}/output-min.js"/>
        <srcfile/>
    </apply>
</target>
```

SampleAntDir/build.xml

该目标包含了一个文件 output.js，由构建过程生成的，并传递给 YUI 压缩器。输出文件指定为同一目录下的 output-min.js。这里假设 yuicompressor.jar 属性包含了 YUI 压缩器的 jar 文件的位置。然后可以使用以下命令运行这个目标：

```
ant js.compress
```

所有的 JavaScript 文件在部署到生产环境之前，都应该使用 YUI 压缩器或者类似的工具进行压缩。给构建过程添加一个压缩 JavaScript 文件的环节以确保每次都进行这个操作。

2. HTTP 压缩

配重指的是实际从服务器传送到浏览器的字节数。因为现在的服务器和浏览器都有压缩功能，这个字节数不一定和代码长度一样。所有的五大 Web 浏览器（IE、Firefox、Safari、Chrome 和 Opera）都支持对所接收的资源进行客户端解压缩。这样服务器端就可以使用服务器端相关功能来压缩 JavaScript 文件。一个指定了文件使用了给定格式进行了压缩的 HTTP 头包含在了服务器响应中。接着浏览器会查看该 HTTP 头确定文件是否已被压缩，然后使用合适的格式进行解压缩。结果是和原来的代码量相比在网络中传递的字节数量大大减少了。

对于 Apache Web 服务器，有两个模块可以进行 HTTP 压缩：mod_gzip（Apache1.3.x）和 mod_deflate（Apache 2.0.x）。对于 mod_gzip，可以给 httpd.conf 文件或者是 .htaccess 文件添加以下代码启用对 JavaScript 的自动压缩：

```
#告诉 mod_zip 要包含任何以 .js 结尾的文件
mod_gzip_item_include          file      \.js$
```

该行代码告诉 mod_zip 要包含来自浏览器请求的任何以 .js 结尾的文件。假设你所有的 JavaScript 文件都以 .js 结尾，就可以压缩所有请求并应用合适的 HTTP 头以表示内容已被压缩。关于 mod_zip 的更多信息，请访问项目网站 http://www.sourceforge.net/projects/mod-gzip/。

对于 mod_deflate，可以类似添加一行代码以保证 JavaScript 文件在被发送之前已被压缩。将以下这一行代码添加到 httpd.conf 文件或者是 .htaccess 文件中：

```
#告诉 mod_deflate 要包含所有的 JavaScript 文件
AddOutputFilterByType DEFLATE application/x-javascript
```

注意这一行代码用到了响应的 MIME 类型来确定是否对其进行压缩。记住虽然<script>的 type

属性用的是 text/javascript，但是 JavaScript 文件一般还是用 application/x-javascript 作为其服务的 MIME 类型。关于 mod_deflate 的更多信息，请访问 http://httpd.apache.org/docs/2.0/mod/mod_deflate.html。

mod_gzip 和 mod_deflate 都可以节省大约 70%的 JavaScript 文件大小。这很大程度上是因为 JavaScript 都是文本文件，因此可以非常有效地进行压缩。减少文件的配重可以减少需要传输到浏览器的时间。记住有一点点细微的代价，因为服务器必须花时间对每个请求压缩文件，当浏览器接收到这些文件后也需要花一些时间解压缩。不过，一般来说，这个代价还是值得的。

大部分 Web 服务器，开源的或是商业的，都有一些 HTTP 压缩功能。请查看服务器的文档说明以确定如何合适地配置压缩。

24.4 小结

随着 JavaScript 开发的成熟，也出现了很多最佳实践。过去一度认为只是一种爱好的东西现在变成了正当的职业，同时还需要经历过去其他编程语言要做的一些研究，如可维护性、性能和部署。

JavaScript 中的可维护性部分涉及到下面的代码约定。

- 来自其他语言中的代码约定可以用于决定何时进行注释，以及如何进行缩进，不过 JavaScript 需要针对其松散类型的性质创造一些特殊的约定。
- 由于 JavaScript 必须与 HTML 和 CSS 共存，所以让各自完全定义其自己的目的非常重要：JavaScript 应该定义行为，HTML 应该定义内容，CSS 应该定义外观。
- 这些职责的混淆会导致难以调试的错误和维护上的问题。

随着 Web 应用中的 JavaScript 数量的增加，性能变得更加重要，因此，你需要牢记以下事项。

- JavaScript 执行所花费的时间直接影响到整个 Web 页面的性能，所以其重要性是不能忽略的。
- 针对基于 C 的语言的很多性能的建议也适用于 JavaScript，如有关循环性能和使用 switch 语句替代 if 语句。
- 还有一个要记住的重要事情，即 DOM 交互开销很大，所以需要限制 DOM 操作的次数。

流程的最后一步是部署。本章讨论了以下一些关键点。

- 为了协助部署，推荐设置一个可以将 JavaScript 合并为较少文件（理想情况是一个）的构建过程。
- 有了构建过程也可以对源代码自动运行额外的处理和过滤。例如，你可以运行 JavaScript 验证器来确保没有语法错误或者是代码没有潜在的问题。
- 在部署前推荐使用压缩器将文件尽可能变小。
- 和 HTTP 压缩一起使用可以让 JavaScript 文件尽可能小，因此对整体页面性能的影响也会最小。