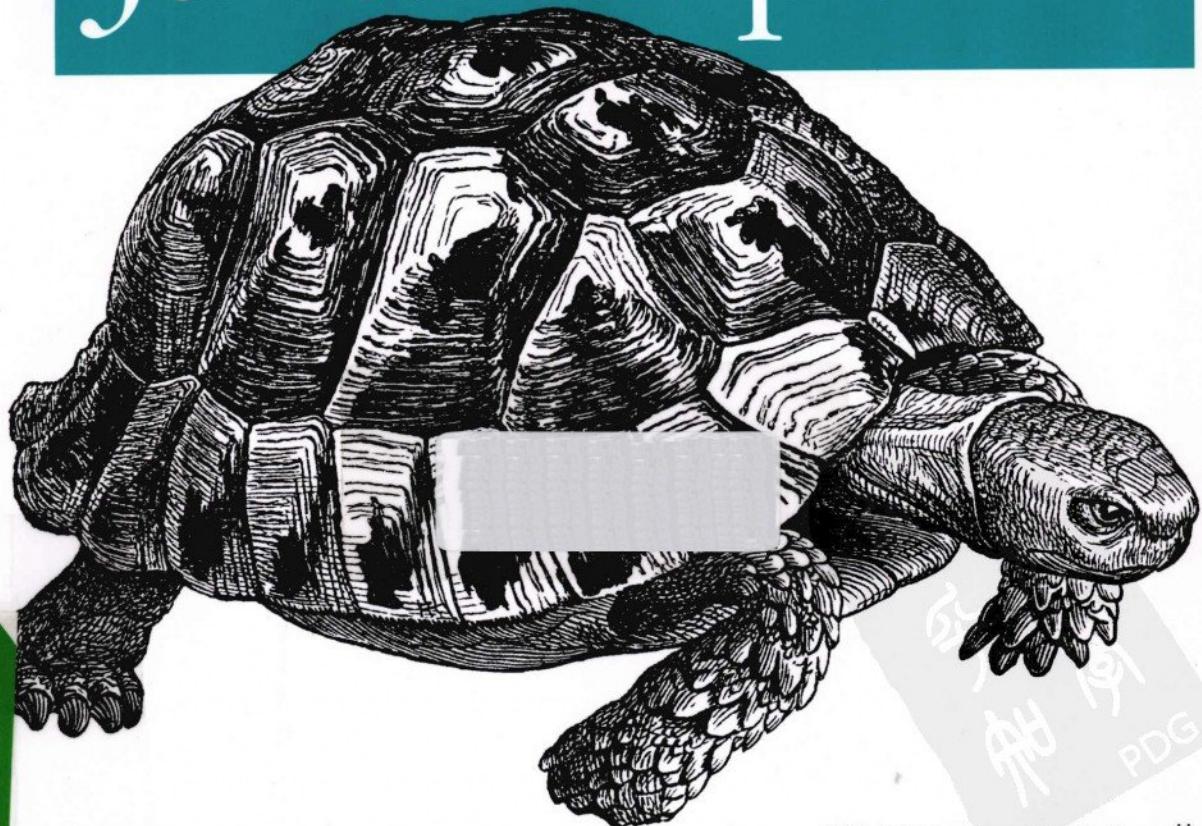


Maintainable JavaScript

编写可维护的 JavaScript



[美] Nicholas C. Zakas 著
李晶 郭凯 张散集 译

O'REILLY®

 人民邮电出版社
POSTS & TELECOM PRESS

编写可维护的JavaScript

每个人都有固定的一套编码习惯，但在团队协作过程中，则需要每个人都遵守统一的编码约定和编程方法。本书介绍如何在团队协作中保持高效的编码，书中的最佳实践包括代码风格、编程技巧以及自动化。你将学到如何写出具有高可维护性的代码，以便团队任何人都可以轻松地理解、修改或扩展你的代码。

本书作者是大名鼎鼎的Nicholas C. Zakas。他是Yahoo!的首席前端开发工程师。在完成了从一名“独行侠”到“团队精英”的蜕变后，他站在前端工程师的角度，提炼出了众多的最佳编程实践，其中包括很多其他业内权威所推崇的最佳法则。应用这些技巧和技术，你的团队编程可以从狭义的个人偏好的阴霾中走出来，走向真正的高效能和高水准。

本书包括以下内容：

- 为团队开发建立特定的编程约定；
- 使用工具（如JSLint和JSHint）让团队开发走向正轨；
- 构建编码风格手册（如基本的格式化），帮助开发团队从“游击队”走向“正规军”；
- 应用编程实践来解决常见问题，提高编码质量；
- 使用多种工具创建JavaScript自动化构建系统；
- 使用诸如YUI Test Selenium引擎等工具来集成基于浏览器的JavaScript测试。

Nicholas C. Zakas是一名前端开发顾问、作者、演讲家。他在Yahoo!供职超过5年时间。在这期间，他曾是Yahoo!首页首席前端工程师和YUI代码贡献者。他著有《JavaScript高级程序设计》、《Ajax高级程序设计》和《高性能JavaScript》。他的博客地址是：<http://www.nczonline.net/>。他的Twitter是：@slicknet。

封面设计：Karen Montgomery, 张健

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行

This Authorized Edition for sale only in the territory of People's Republic of China
(excluding Hong Kong, Macao and Taiwan)

分类建议：计算机/程序设计

人民邮电出版社网址：www.ptpress.com.cn



“本书是一本教你写出具有前瞻性的JavaScript代码的完全手册，在团队作战中特别有用。”

—— Ryan Grove,
Yahoo! YUI工程师

“作者将他十多年工作经验的精华部分浓缩至这本通俗易读的书中。我建议每个开发工程师和在校学生尽早阅读本书。不管你有多少经验，本书中的每一页内容都会让你变得更加优秀且倍受大家欢迎。”

——Lea Verou,
Web设计师兼Web开发人员

“这是一本前端工程师的指南，指明了在编码过程中需要注意的方方面面。提高可维护性是一个非常大的话题，而这本书，是一个非常不错的起点。”

——王保平（玉伯），
支付宝Web前端工程师



ISBN 978-7-115-31008-8



ISBN 978-7-115-31008-8

定价：55.00 元

O'REILLY®

编写可维护的 JavaScript

[美] Nicholas C. Zakas 著

李 晶 郭 凯 张散集 译

人民邮电出版社

北京



图书在版编目（C I P）数据

编写可维护的JavaScript / (美) 扎卡斯著
(Zakas, N. C.) 著 ; 李晶, 郭凯, 张散集译. — 北京 :
人民邮电出版社, 2013. 4
ISBN 978-7-115-31008-8

I. ①编… II. ①扎… ②李… ③郭… ④张… III.
①JAVA语言—程序设计 IV. ①TP312

中国版本图书馆CIP数据核字(2013)第025073号

版权声明

Copyright © 2012 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Posts & Telecom Press, 2013. Authorized translation of the English edition, 2012 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

本书中文简体字版由 O'Reilly Media, Inc. 授权人民邮电出版社出版。未经出版者书面许可，对本书的任何部分不得以任何方式复制或抄袭。

版权所有，侵权必究。

编写可维护的 JavaScript

-
- ◆ 著 [美] Nicholas C. Zakas
 - 译 李晶 郭凯 张散集
 - 责任编辑 陈冀康
 - ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
 - 邮编 100061 电子邮件 315@ptpress.com.cn
 - 网址 <http://www.ptpress.com.cn>
 - 北京隆昌伟业印刷有限公司印刷
 - ◆ 开本：787×1092 1/16
 - 印张：15.5
 - 字数：281 千字 2013 年 4 月第 1 版
 - 印数：1—3 500 册 2013 年 4 月北京第 1 次印刷

著作权合同登记号 图字：01-2012-8552 号

ISBN 978-7-115-31008-8

定价：55.00 元

读者服务热线：(010) 67132692 印装质量热线：(010) 67129223

反盗版热线：(010) 67171154

广告经营许可证：京崇工商广字第 0021 号

PDG

目录

第一部分 编程风格	1
第1章 基本的格式化	4
1.1 缩进层级	4
1.2 语句结尾	7
1.3 行的长度	8
1.4 换行	9
1.5 空行	10
1.6 命名	11
1.6.1 变量和函数	12
1.6.2 常量	13
1.6.3 构造函数	14
1.7 直接量	15
1.7.1 字符串	15
1.7.2 数字	16
1.7.3 null	17
1.7.4 undefined	18
1.7.5 对象直接量	19
1.7.6 数组直接量	20
第2章 注释	21
2.1 单行注释	21
2.2 多行注释	23
2.3 使用注释	24
2.3.1 难于理解的代码	25
2.3.2 可能被误认为错误的代码	26
2.3.3 浏览器特性 hack	26
2.4 文档注释	27
第3章 语句和表达式	30
3.1 花括号的对齐方式	31

3.2 块语句间隔	32
3.3 switch 语句	33
3.3.1 缩进	33
3.3.2 case 语句的“连续执行”	35
3.3.3 default	36
3.4 with 语句	37
3.5 for 循环	37
3.6 for-in 循环	39
第 4 章 变量、函数和运算符	41
4.1 变量声明	41
4.2 函数声明	44
4.3 函数调用间隔	45
4.4 立即调用的函数	46
4.5 严格模式	47
4.6 相等	49
4.6.1 eval()	51
4.6.2 原始包装类型	52
第二部分 编程实践	54
第 5 章 UI 层的松耦合	55
5.1 什么是松耦合	56
5.2 将 JavaScript 从 CSS 中抽离	57
5.3 将 CSS 从 JavaScript 中抽离	58
5.4 将 JavaScript 从 HTML 中抽离	60
5.5 将 HTML 从 JavaScript 中抽离	62
5.5.1 方法 1：从服务器加载	63
5.5.2 方法 2：简单客户端模板	64
5.5.3 方法 3：复杂客户端模板	67
第 6 章 避免使用全局变量	70
6.1 全局变量带来的问题	70
6.1.1 命名冲突	71
6.1.2 代码的脆弱性	71
6.1.3 难以测试	72

6.2	意外的全局变量	72
避免意外的全局变量	73	
6.3	单全局变量方式	74
6.3.1	命名空间	76
6.3.2	模块	78
6.4	零全局变量	81
第 7 章	事件处理	83
7.1	典型用法	83
7.2	规则 1：隔离应用逻辑	84
7.3	规则 2：不要分发事件对象	85
第 8 章	避免“空比较”	88
8.1	检测原始值	88
8.2	检测引用值	90
8.2.1	检测函数	92
8.2.2	检测数组	94
8.3	检测属性	95
第 9 章	将配置数据从代码中分离出来	98
9.1	什么是配置数据	98
9.2	抽离配置数据	99
9.3	保存配置数据	100
第 10 章	抛出自定义错误	103
10.1	错误的本质	103
10.2	在 JavaScript 中抛出错误	104
10.3	抛出错误的好处	105
10.4	何时抛出错误	106
10.5	try-catch 语句	107
10.6	错误类型	109
第 11 章	不是你的对象不要动	112
11.1	什么是你的	112
11.2	原则	113
11.2.1	不覆盖方法	113
11.2.2	不新增方法	114

11.2.3 不删除方法	116
11.3 更好的途径	117
11.3.1 基于对象的继承	118
11.3.2 基于类型的继承	119
11.3.3 门面模式	120
11.4 关于 Polyfill 的注解	121
11.5 阻止修改	122
第 12 章 浏览器嗅探	125
12.1 User-Agent 检测	125
12.2 特性检测	127
12.3 避免特性推断	129
12.4 避免浏览器推断	130
12.5 应当如何取舍	134
第三部分 自动化	135
第 13 章 文件和目录结构	137
13.1 最佳实践	137
13.2 基本结构	138
第 14 章 Ant	143
14.1 安装	143
14.2 配置文件	143
14.3 执行构建	145
14.4 目标操作的依赖	145
14.5 属性	146
14.6 Buildr 项目	148
第 15 章 校验	149
15.1 查找文件	149
15.2 任务	150
15.3 增强的目标操作	152
15.4 其他方面的改进	153
15.5 Buildr 任务	154
第 16 章 文件合并和加工	156
16.1 任务	156

16.2 行尾结束符	157
16.3 文件头和文件尾	158
16.4 加工文件	159
第 17 章 文件精简和压缩	163
17.1 文件精简	163
17.1.1 使用 YUI Compressor 精简代码	165
17.1.2 用 Closure Compiler 精简	167
17.1.3 使用 UglifyJS 精简	169
17.2 压缩	170
17.2.1 运行时压缩	171
17.2.2 构建时压缩	171
第 18 章 文档化	175
18.1 JSDoc Toolkit	175
18.2 YUI Doc	177
第 19 章 自动化测试	180
19.1 YUI Test Selenium 引擎	180
19.1.1 配置一台 Selenium 服务器	181
19.1.2 配置 YUI Test Selenium 引擎	181
19.1.3 使用 YUI Test Selenium 引擎	181
19.1.4 Ant 的配置写法	183
19.2 Yeti	184
19.3 PhantomJS	186
19.3.1 安装及使用	186
19.3.2 Ant 的配置写法	187
19.4 JsTestDriver	188
19.4.1 安装及使用	188
19.4.2 Ant 的配置写法	189
第 20 章 组装到一起	191
20.1 被忽略的细节	191
20.2 编制打包计划	192
20.2.1 开发版本的构建	193
20.2.2 集成版本的构建	194

20.2.3	发布版本的构建	195
20.3	使用 CI 系统	196
20.3.1	Jenkins	196
20.3.2	其他 CI 系统	199
附录 A	JavaScript 编码风格指南	200
附录 B	JavaScript 工具集	223



第一部分

编程风格

“程序是写给人读的，只是偶尔让计算机执行一下。”

——Donald Knuth。^①

当你刚刚组建一个团队时，团队中的每个人都各自有一套编程习惯。毕竟，每个成员都有着不同的背景。有些人可能来自某个“皮包公司”(one-man shop)，身兼数职，在公司里什么事都做；还有些人会来自不同的团队，对某种特定的做事风格情有独钟（或恨之入骨）。每个人都觉得代码应当按照自己的想法来写，这些通常被归纳为个人编程嗜好。在这个过程中^②应当尽早将确定统一的编程风格纳入议题。



我们会经常碰到这两个术语：“编程风格”(style guideline)和“编码规范”(code convention)。编程风格是编码规范的一种，用来规约单文件中代码的规划。编码规范还包含编程最佳实践、文件和目录的规划以及注释等方面。本书集中讨论JavaScript的编码规范。

为什么要讨论编程风格

提炼编程风格是一道工序，花再多的时间也不为过。毕竟每个人都有自己的想法，

① 译注：高德纳（Donald Ervin Knuth）是世界顶级计算机科学家之一，被公认为现代计算机科学的鼻祖，著有《计算机程序设计艺术》(The Art of Computer Programming)等经典著作，在不多的业余时间里，Knuth不仅写小说，还是一位音乐家、作曲家、管风琴设计师。

② 译注：意指组建团队的过程。

如果一天当中你有 8 小时是在写代码，那么你自然希望用一种舒服的方式来写代码。刚开始，团队成员对新的编程风格有点不适应，全靠强势的项目组长强制推行才得以持续。一旦风格确立后，这套编程风格就会促成团队成员高水准的协作，因为所有代码（的风格）看起来极为类似。

在团队开发中，所有的代码看起来风格一致是极其重要的，原因有以下几点。

- 任何开发者都不会在乎某个文件的作者是谁，也没有必要花费额外精力去理解代码逻辑并重新排版，因为所有代码排版格式看起来非常一致。我们打开一个文件时所干的第一件事，常常不是立即开始工作而是首先修复代码的缩进，当项目很庞大时，你会体会到统一的编程风格的确大幅度节省了时间成本。
- 我能很容易地识别出问题代码并发现错误。如果所有代码看起来很像，当你看到一段与众不同的代码时，很可能错误就产生在这段代码中。

毫无疑问，全球性的大公司都对外或对内发布过编程风格文档。

编程风格是个人的事情，只有放到团队开发中才能发挥作用。本书的这部分给出了 JavaScript 编码规范中值得关注（推荐）的方面。在某些场景中，很难说哪种编程风格好，哪种编程风格不好，因为有些编程风格只是某些人的偏好。本章不是向你灌输我个人的风格偏好，而是提炼出了编程风格应当遵循的重要的通用准则。本书附录 A 中给出了我个人的 JavaScript 编程风格。

有用的工具

开发编码指南是一件非常困难的事情——执行是另外一回事。在团队中通过讨论达成一致和进行代码评审（code review）时，每个人都很关注编码风格，但在平时大家却常常将这些抛在脑后。工具可以对每个人实时跟踪。这里有两个用来检查编程风格的工具，这两个工具非常有用：JSLint 和 JSHint。

JSLint 是由 Douglas Crockford 创建的。这是一个通用的 JavaScript 代码质量检查工具。最开始，JSLint 只是一个简单的查找不符合 JavaScript 模式的、错误的小工具。经过数年的进化，JSLint 已经成为一个有用的工具，不仅仅可以找出代码中潜在的错误，而且能针对你的代码给出编码风格上的警告。

Crockford 将他对 JavaScript 风格的观点分成了三个不同的部分。

- “JavaScript 风格的组成部分（第一部分）” (<http://javascript.crockford.com/style1.html>)，包含基本的模式和语法。
- “JavaScript 风格的组成部分（第二部分）” (<http://javascript.crockford.com/style2.html>)，包含一般性的 JavaScript 惯用法。
- “JavaScript 编程语言的编码规范” (<http://javascript.crockford.com/code.html>)，这个规范更加全面，从前两部分中提炼出了编程风格的精华部分，同时增补了少量的编程风格指引。

JSLint 直接吸纳了很多 Crockford 所提炼的编程风格，而且很多时候我们无法关闭 JSLint 中检查编程风格的功能。所以 JSLint 是一个非常棒的工具，当然前提是你可以认可 Crockford 关于编程风格的观点。

JSHint 是 JSLint 的一个分支项目，由 Anton Kovalyov 创建并维护。JSHint 的目标是提供更加个性化的 JavaScript 代码质量和编程风格检查的工具。比如，当出现语法错误的时候，JSHint 几乎可以关掉所有编程风格检查，这样你可以完全自定义消息提示。Kovalyov 非常鼓励大家通过 GitHub (<http://github.com>) 上的源代码库参与 JSHint 项目并为之贡献代码。

你可以将这些工具中的一种集成到打包过程中，通过这种方式推行编码规范是一个不错的方法。这种方法同时可以监控你的 JavaScript 代码中潜在的错误。

第1章

基本的格式化

编程风格指南的核心是基本的格式化规则（formatting rule）。这些规则直接决定了如何编写高水准的代码。与在学校学习写字时所用的方格纸类似，基本的格式化规则将指引开发者以特定的风格编写代码。这些规则通常包含一些你不太在意的有关语法的信息，但对于编写清晰连贯的代码段来说，每一条信息都是非常重要的。

1.1 缩进层级

关于 JavaScript 编码风格，我们首先要讨论的是（几乎所有的语言都是如此）如何处理缩进。对这个话题是可以争论上好几个小时的，缩进甚至关系到软件工程师的价值观。在确定编程风格之初应当首先确定缩进格式，这非常重要，以免工程师后续会陷入那个老生常谈的打开文件时二话不说先重排代码缩进的问题之中。来看一下这段代码（为了演示，这里故意修改了示例代码的缩进）。

```
if (wl && wl.length) {
    for (i = 0, l = wl.length; i < l; ++i) {
        p = wl[i];
        type = Y.Lang.type(r[p]);
        if (s.hasOwnProperty(p)) { if (merge && type == 'object') {

            Y.mix(r[p], s[p]);
        } else if (ov || !(p in r)) {
            r[p] = s[p];
        }
    }
}
```

```
}
```

快速读懂这段代码不容易。这里的缩进并不统一，一眼看去 `else` 是对应到第 1 行的 `if` 语句。但实际上这个 `else` 和代码第 5 行的 `if` 语句相对应。罪魁祸首是多位开发人员在同一段代码里应用了不同的缩进风格。这恰恰说明了统一缩进风格的重要性。如果有适当的缩进，这段代码将变得更加易读。

```
if (wl && wl.length) {
    for (i = 0, l = wl.length; i < l; ++i) {
        p = wl[i];
        type = Y.Lang.type(r[p]);
        if (s.hasOwnProperty(p)) {
            if (merge && type == 'object') {
                Y.mix(r[p], s[p]);
            } else if (ov || !(p in r)) {
                r[p] = s[p];
            }
        }
    }
}
```

坚持使用适度的缩进是万里长征的第一步——本章在下面将提到这种做法可以带来其他可维护性方面的提升。

对于大多数编程风格来说，代码到底应该如何缩进并没有统一的共识。有两种主张。

使用制表符进行缩进

每一个缩进层级都用单独的制表符 (tab character) 表示。所以一个缩进层级是一个制表符，两个缩进层级为两个制表符，以此类推。这种方法有两个主要的好处。第一，制表符和缩进层级之间是一对一的关系，这是符合逻辑的。第二，文本编辑器可以配置制表符的展现长度^①，因此那些想修改缩进尺寸的开发者可以通过配置文本编辑器来实现，想长即长，想短可短。使用制表符作缩进的主要缺点是，系统对制表符的解释不一致。你会发觉在某个系统中用一款编辑器打开文件时看到的缩进，和在另外一个系统中用相同的编辑器打开文件时看到的不一样。对于那些追求（代码展现）一致性的开发者来说，这会带来一些困惑。这些差异、争论会导致不同的开发者对同一段代码有不同的看法的，而这些正是团队开发需要规避的。

^① 译注：通常一个制表符长度相当于 4 个字符。

使用空格符进行缩进

每个缩进层级由多个空格字符组成。在这种观点中有三种具体的做法：2个空格表示一个缩进，2个空格表示一个缩进，以及8个空格表示一个缩进。这三种做法在其他很多编程语言中都能找到渊源。实际上，很多团队选择4个空格的缩进，对于那些习惯用2个空格缩进和用8个空格缩进的人来说，4个空格缩进是一种折中的选择。使用空格作缩进的好处是，在所有的系统和编辑器中，文件的展现格式不会有任何差异。可以在文本编辑器中配置敲击Tab键时插入几个空格。也就是说所有开发者都可以看到一模一样的代码呈现。使用空格缩进的缺点是，对于单个开发者来说，使用一个没有配置好的文本编辑器创建格式化的代码的方式非常原始。

尽管有人争辩说应当优先考虑使用一种缩进约定，但说到底这只是一个团队偏好的问题。这里我们给出一些各式各样的缩进风格作为参考。

- jQuery 核心风格指南（jQuery Core Style Guide）明确规定使用制表符缩进。
- Douglas Crockford 的 JavaScript 代码规范（Douglas Crockford's Code Conventions for the JavaScript Programming Language）规定使用4个空格字符的缩进。
- SproutCore 风格指南（SproutCore Style Guide）规定使用2个空格的缩进。
- Google 的 JavaScript 风格指南（Google JavaScript Style Guide）规定使用2个空格的缩进。
- Dojo 编程风格指南（Dojo Style Guide）规定使用制表符缩进。

我推荐使用4个空格字符为一个缩进层级。很多文本编辑器都默认将缩进设置为4个空格，你可以在编辑器中配置敲入Tab键时插入4个空格。使用2个空格的缩进时，代码的视觉展现并不是最优的，至少看起来是这样。

尽管是选择制表符还是选择空格做缩进只是一种个人偏好，但绝对不要将两者混用，这非常重要。这么做会导致文件的格式很糟糕，而且需要做不少清理工作，就像本节的第一段示例代码显示的那样。

1.2 语句结尾

有一件很有意思且很容易让人困惑的事情，那就是 JavaScript 的语句要么独占一行，要么以分号结尾。类似 C 的编程语言，诸如 C++ 和 Java，都采用这种行结束写法，即结尾使用分号。下面这两段示例代码都是合法的 JavaScript。

```
// 合法的代码
var name = "Nicholas";
function sayName() {
    alert(name);
}

// 合法的代码，但不推荐这样写
var name = "Nicholas"
function sayName() {
    alert(name)
}
```

有赖于分析器的自动分号插入（Automatic Semicolon Insertion, ASI）机制，JavaScript 代码省略分号也是可以正常工作的。ASI 会自动寻找代码中应当使用分号但实际上没有分号的位置，并插入分号。大多数场景下 ASI 都会正确插入分号，不会产生错误。但 ASI 的分号插入规则非常复杂且很难记住，因此我推荐不要省略分号。看一下这段代码。

```
// 原始代码
function getData() {
    return
    {
        title: "Maintainable JavaScript",
        author: "Nicholas C. Zakas"
    }
}
// 分析器会将它理解成
function getData() {
    return;
{
    title: "Maintainable JavaScript",
    author: "Nicholas C. Zakas"
};
}
```

在这段代码中，函数 `getData()` 的本意是返回一个包含一些数据的对象。然而，`return` 之后新起了一行，导致 `return` 后被插入了一个分号，这会导致函数返回值是 `undefined`。

可以通过将左花括号移至与 `return` 同一行的位置来修复这个问题。

```
// 这段代码工作正常，尽管没有用分号
function getData() {
    return {
        title: "Movable Type",
        author: "Nicholas C. Zakas"
    }
}
```

ASI 在某些场景下是很管用的，特别是，有时候 ASI 可以帮助减少代码错误。当某个场景我们认为不需要插入分号而 ASI 认为需要插入时，常常会产生错误。我发现很多开发人员，尤其是新手们，更倾向于使用分号而不是省略它们。

Douglas Crockford 针对 JavaScript 提炼出的编程规范（下文统称为 Crockford 的编程规范）推荐总是使用分号，同样，jQuery 核心风格指南、Google 的 JavaScript 风格指南以及 Dojo 编程风格指南都推荐不要省略分号。如果省略了分号，JSLint 和 JSHint 默认都会有警告。

1.3 行的长度

和缩进话题息息相关的是行的长度。如果一行代码太长，编辑窗口出现了横向滚动条，会让开发人员感觉很别扭。即便是在当今的宽屏显示器中，保持合适的代码行长度也会极大地提高工程师的生产力。很多语言的编程规范都提到一行代码最长不应当超过 80 个字符。这个数值来源于很久之前文本编辑器的单行最多字符限制，即编辑器中单行最多只能显示 80 个字符，超过 80 个字符的行要么折行，要么被隐藏起来，这些都是我们所不希望的。相比 20 年前的编辑器，现在的文本编辑器更加精巧，但仍然有很多编辑器保留了单行 80 个字符的限制。此外关于行长度，还有一些常见的建议。

1. Java 语言编程规范中规定源码里单行长度不超过 80 个字符，文档中代码单行长度不超过 70 个字符。
2. Android 开发者编码风格指南规定单行代码长度不超过 100 个字符。
3. 非官方的 Ruby 编程规范中规定单行代码长度不超过 80 个字符。

4. Python 编程规范中规定单行代码长度不超过 79 个字符。

Java Script 风格指南中很少提及行的长度，但 Crockford 的代码规范中指定一行的长度为 80 个字符。我也倾向于将行长度限定在 80 个字符。

1.4 换行

当一行长度达到了单行最大字符数限制时，就需要手动将一行拆成两行。通常我们会在运算符后换行，下一行会增加两个层级的缩进。比如（假定缩进为 4 个字符）下面这样。

```
// 好的做法：在运算符后换行，第二行追加两个缩进
callAFunction(document, element, window, "some string value", true, 123,
               navigator);

// 不好的做法：第二行只有一个缩进
callAFunction(document, element, window, "some string value", true, 123,
               navigator);

// 不好的做法：在运算符之前换行了
callAFunction(document, element, window, "some string value", true, 123
               , navigator);
```

在这个例子中，逗号是一个运算符，应当作为前一行的行尾。这个换行位置非常重要，因为 ASI 机制会在某些场景下在行结束的位置插入分号。总是将一个运算符置于行尾，ASI 就不会自作主张地插入分号，也就避免了错误的发生。

对于语句来说，同样也可以应用下面这种换行规则。

```
if (isLeapYear && isFebruary && day == 29 && itsYourBirthday &&
    noPlans) {

    waitAnotherFourYears();
}
```

在这段代码中，if 条件语句被拆分成了两行，断行在`&&`运算符之后。需要注意的是，if 语句的主体部分依然是一个缩进，这样更容易阅读。

这个规则有一个例外：当给变量赋值时，第二行的位置应当和赋值运算符的位置保持对齐。比如：

```
var result = something + anotherThing + yetAnotherThing + somethingElse +
anotherSomethingElse;
```

这段代码里，变量 `anotherSomethingElse` 和首行的 `something` 保持左对齐，确保代码的可读性，并能一眼看清楚折行文本的上下文。

1.5 空行

在编程规范中，空行是常常被忽略的一个方面。通常来讲，代码看起来应当像一系列可读的段落，而不是一大段揉在一起的连续文本。有时一段代码的语义和另一段代码不相关，这时就应该使用空行将它们分隔，确保语义有关联的代码展现在一起。我们可以为 1.1 节里的示例代码加入一些空行，以更好地提升代码的可读性，下面是最初的代码：

```
if (wl && wl.length) {
    for (i = 0, l = wl.length; i < l; ++i) {
        p = wl[i];
        type = Y.Lang.type(r[p]);
        if (s.hasOwnProperty(p)) {
            if (merge && type == 'object') {
                Y.mix(r[p], s[p]);
            } else if (ov || !(p in r)) {
                r[p] = s[p];
            }
        }
    }
}
```

给这段代码添加了几个空行之后，得到：

```
if (wl && wl.length) {

    for (i = 0, l = wl.length; i < l; ++i) {
        p = wl[i];
        type = Y.Lang.type(r[p]);

        if (s.hasOwnProperty(p)) {

            if (merge && type == 'object') {
                Y.mix(r[p], s[p]);
            } else if (ov || !(p in r)) {
                r[p] = s[p];
            }
        }
    }
}
```

```
    }  
}  
}
```

这段示例代码中所展示的编程规范是在每个流控制语句之前（比如 if 和 for 语句）添加空行。这样做能使你更流畅地阅读这些语句。一般来讲，在下面这些场景中添加空行也是不错的主意。

- 在方法之间。
- 在方法中的局部变量（local variable）和第一条语句之间。
- 在多行或单行注释之前。
- 在方法内的逻辑片段之间插入空行，提高可读性。

但并没有一个编程规范对空行的使用给出任何具体建议，Crockford 的编程规范也只提到要审慎地使用空行。

1.6 命名

“计算机科学只存在两个难题：缓存失效和命名。” —— Phil Karlton。

只要是写代码，都会涉及变量和函数，因此变量和函数命名对于增强代码可读性至关重要。JavaScript 语言的核心 ECMAScript，即是遵照了驼峰式大小写（Camel case）^① 命名法。驼峰式大小写（Camel Case）命名法是由小写字母开始的，后续每个单词首字母都大写，比如：

```
var thisIsMyName;  
var anotherVariable;  
var aVeryLongVariableName;
```

一般来讲，你应当遵循你使用的语言核心所采用的命名规范，因此大部分 JavaScript 程序员使用驼峰命名法来给变量和函数命名。Google 的 JavaScript 风格指南、SproutCore 编程风格指南以及 Dojo 编程风格指南在大部分场景中也都采用了小驼峰（Camel

^① 译注：Camel Case 和 Pascal Case 都翻译成“驼峰式大小写”，但两者含义有所不同，Camel Case 包括“小驼峰式”和“大驼峰式”，在本章中，Camel Case 被作者用来特指“小驼峰式大小写”（即首字母小写）命名法，Pascal Case 则特指“大驼峰式大小写”（即首字母大写）命名法，请读者留意。

Case) 命名。

尽管小驼峰 (Camel Case) 命名法是最常见的命名方法，但我们不排斥更多其他的命名风格。

在 2000 年左右，JavaScript 中流行另外一种命名方法——匈牙利命名法。这种命名方法的特点是，名字之前冠以类型标识符前缀，比如 `sName` 表示字符串，`iCount` 表示整数。这种风格已经是明日黄花风光不再了，当前主流的编程规范都不推荐这种命名法。

1.6.1 变量和函数

变量名应当总是遵守驼峰大小写命名法，并且命名前缀应当是名词。以名词作为前缀可以让变量和函数区分开来，因为函数名前缀应当是动词。这里有一些例子。

```
// 好的写法
var count = 10;
var myName = "Nicholas";
var found = true;

// 不好的写法：变量看起来像函数
var getCount = 10;
var isFound = true;

// 好的写法
function getName() {
    return myName;
}

// 不好的写法：函数看起来像变量
function theName() {
    return myName;
}
```

命名不仅是一门科学，更是一门技术，但通常来讲，命名长度应该尽可能短，并抓住要点。尽量在变量名中体现出值的数据类型。比如，命名 `count`、`length` 和 `size` 表明数据类型是数字，而命名 `name`、`title`、和 `message` 表明数据类型是字符串。但用单个字符命名的变量诸如 `i`、`j`、和 `k` 通常在循环中使用。使用这些能够体现出数据类型的命名，可以让你的代码容易被别人和自己读懂。

要避免使用没有意义的命名。那些诸如 `foo`、`bar` 和 `tmp` 之类的命名也应当避免，当然开发者的工具箱中还有很多这样可以随拿随用的名字，但不要让这些命名承载

其他的附加含义。对于其他开发者来说，如果没有看过上下文，是无法理解这些变量的用处的。

对于函数和方法命名来说，第一个单词应该是动词，这里有一些使用动词常见的约定。

动 词	含 义
can	函数返回一个布尔值
has	函数返回一个布尔值
is	函数返回一个布尔值
get	函数返回一个非布尔值
set	函数用来保存一个值

以这些约定作为切入点可以让代码可读性更佳，这里有一些例子。

```
if (isEnabled()) {  
    setName("Nicholas");  
}  
  
if (getName() === "Nicholas") {  
    doSomething();  
}
```

尽管这些函数命名细则并没有被归纳入当下流行的编程风格中，但在很多流行的库中，JavaScript 开发者会发现存在不少这种“伪标准”(pseudostandard)。

jQuery 显然并没有遵循这种函数命名约定，一部分原因在于 jQuery 中方法的使用方式，很多方法同时用作 getter 和 setter。比如，`$(“body”).attr(“class”)`可以取到 class 属性的值，而`$(“body”).attr(“class”, “selected”)`可以给 class 属性赋值。尽管如此，我还是推荐使用动词作为函数名前缀。

1.6.2 常量

在 ECMAScript 6 之前，JavaScript 中并没有真正的常量的概念。然而，这并不能阻止开发者将变量用作常量。为了区分普通的变量（变量的值是可变的）和常量（常量的值初始化之后就不能变了），一种通用的命名约定应运而生。这个约定源自于 C 语言，它使用大写字母和下划线来命名，下划线用以分隔单词，比如：

```
var MAX_COUNT = 10;  
var URL = "http://www.nczonline.net/";
```

需要注意的是，这里仅仅是应用了不同命名约定的变量而已，因此它们的值都是可以被修改的。使用这种不同的约定来定义普通的变量和常量，使两者非常易于区分。来看一下这段代码：

```
if (count < MAX_COUNT) {  
    doSomething();  
}
```

在这段代码中，一眼就能看出 `count` 是变量，其值是可变的，而 `MAX_COUNT` 表示常量，它的值不会被修改。这个约定为底层代码（underlying code）增加了另外一层语义。

Google 的 JavaScript 风格指南、SproutCore 编程风格指南以及 Dojo 编程风格指南中都提到，要使用这种习惯来命名常量。（Dojo 编程风格指南中也允许使用大驼峰命名法大小写（Pascal Case）来命名常量，接下来会提到）。

1.6.3 构造函数

在 JavaScript 中，构造函数只不过是前面冠以 `new` 运算符的函数，用来创建对象。语言本身已经包含了很多内置构造函数，比如 `Object` 和 `RegExp`，同样开发者也可以创建自己的构造函数来生成新类型。正如其他的命名约定一样，构造函数的命名风格也和本地语言（Native Language）保持一致，因此构造函数的命名遵照大驼峰命名法（Pascal Case）。

Pascal Case 和 Camel Case 都表示“驼峰大小写”，二者的区别在于 Pascal Case 以大写字母开始。因此 `anotherName` 可以替换成 `AnotherName`。这样做可以将构造函数从变量和普通函数中区分出来。构造函数的命名也常常是名词，因为它们是用来创建某个类型的实例的。这里有一些例子：

```
// 好的做法  
function Person(name) {  
    this.name = name;  
}  
  
Person.prototype.sayName = function() {  
    alert(this.name);  
};  
  
var me = new Person("Nicholas");
```

遵守这条约定同样可以帮助我们快速定位问题，这接下来会提到。你知道在以大驼峰命名法（Pascal case）命名的函数如果是名词的话，前面一定会有 new 运算符。看一下这段代码：

```
var me = Person("Nicholas");
var you = getPerson("Michael");
```

这段代码中，根据上文提到的命名约定，我们一眼就可以看出第一行出了问题，但第二行看起来还 ok。

Crockford 的编程规范、Google 的 JavaScript 风格指南以及 Dojo 编程风格指南都推荐这种实践。如果构造函数的首字母不是大写，或者构造函数之前没有 new 运算符，JSLint 都会给出警告。而在 JSHint 中，只有你开启了一个特殊的 newcap 选项后，才会对首字母不是大写的构造函数给出警告。

1.7 直接量

JavaScript 中包含一些类型的原始值：字符串、数字、布尔值、null 和 undefined。同样也包含对象直接量和数组直接量。这其中，只有布尔值是自解释（self-explanatory）的，其他的类型或多或少都需要思考一下它们如何才能更精确地表示出来。

1.7.1 字符串

在 JavaScript 中，字符串是独一无二的。字符串可以用双引号括起来，也可以用单引号括起来。比如：

```
// 合法的 JavaScript 代码
var name = "Nicholas says, \"Hi.\"";

// 也是合法的 JavaScript 代码
var name = 'Nicholas says, "Hi"';
```

和 Java、PHP 这些语言不同，使用单引号括起字符串和双引号括起字符串在功能上并无不同。除了内部出现字符串界定符（string delimiter）时需要转义之外，两种做法在功效上完全一致。因此在这段示例代码中，在使用双引号括起来的字符串里需要对双引号进行转义，而在使用单引号括起来的字符串里则不必如此。你需要关心的是，你的代码应当从头到尾只保持一种风格。

Crockford 的编程规范和 jQuery 核心风格指南都使用双引号来括住字符串。Google 的 JavaScript 风格指南使用单引号括住字符串。我倾向于使用双引号，因为我经常在 Java 和 JavaScript 之间来回切换。由于 Java 只使用双引号括住字符串，我发现如果在 JavaScript 中也使用这个约定，我会很容易在上下文之间相互切换。这类问题应当在制定规范之初就考虑清楚：这样可以最大程度地减轻工程师的开发负担。

关于字符串还有另外一个问题需要注意，即创建多行字符串。这个特性并非来自 JavaScript 语言本身，却在几乎所有的（JavaScript）引擎中正常工作。

```
// 不好的写法
var longString = "Here's the story, of a man \
named Brady.";
```

尽管从技术上讲这种写法是非法的 JavaScript 语法，但的确能在代码中创建多行字符串。通常不推荐使用这种写法，因为它是一种奇技淫巧而非语言特性，并且在 Google 的 JavaScript 风格指南中是明确禁止的。多行字符串的一种替代写法是，使用字符串连接符 (+) 将字符串分成多份。

```
// Good
var longString = "Here's the story, of a man " +
    "named Brady.;"
```

1.7.2 数字

在 JavaScript 中的数字类型只有一种，因为所有数字形式——整数和浮点数——都存储为相同的数据类型。还有一些其他的数字直接量格式来表示不同的数据格式。其中大部分写法都很好用，但也有一些写法有问题。

```
// 整数
var count = 10;

// 小数
var price = 10.0;
var price = 10.00;

// 不推荐的小数写法：没有小数部分
var price = 10.;

// 不推荐的小数写法：没有整数部分
```

```
var price = .1;  
  
// 不推荐的写法：八进制写法已经被弃用了  
var num = 010;  
  
// 十六进制写法  
var num = 0xA2;  
  
// 科学计数法  
var num = 1e23;
```

前两种有问题的写法分别是省略了小数部分，比如 10.，和省略了整数部分，比如.1。每种写法都有同一个问题：很难搞清楚被省略小数点之前或之后的部分是不小心丢掉了还是刻意为之。很可能是开发者不小心漏掉了。因此为了避免歧义，请不要省略小数点之前或之后的数字。Dojo 编程风格指南明确禁止这两种写法。JSLint 和 JSHint 对这两种写法都会给出警告。

最后一个有问题的写法是八进制数字写法。长久以来，JavaScript 支持八进制数字写法是很多错误和歧义的根源。数字直接量 010 不是表示 10，而是表示八进制中的 8。大多数开发者对八进制格式并不熟悉，也很少用到，所以最好的做法是在代码中禁止八进制直接量。尽管在所有流行的编程规范中没有关于此的规定，但在 JSlint 和 JSHint 中都会对八进制直接量给出警告。

1.7.3 null

null 是一个特殊值，但我们常常误解它，将它和 undefined 搞混。在下列场景中应当使用 null。

- 用来初始化一个变量，这个变量可能赋值为一个对象。
- 用来和一个已经初始化的变量比较，这个变量可以是也可以不是一个对象。
- 当函数的参数期望是对象时，用作参数传入。
- 当函数的返回值期望是对象时，用作返回值传出。

还有下面一些场景不应当使用 null。

- 不要使用 null 来检测是否传入了某个参数。

- 不要用 null 来检测一个未初始化的变量。

这里有一些示例代码。

```
// 好的用法
var person = null;

// 好的用法
function getPerson() {
    if (condition) {
        return new Person("Nicholas");
    } else {
        return null;
    }
}

// 好的用法
var person = getPerson();
if (person !== null) {
    doSomething();
}

// 不好的写法：用来和未初始化的变量比较
var person;
if (person != null) {
    doSomething();
}

// 不好的写法：检测是否传入了参数
function doSomething(arg1, arg2, arg3, arg4) {
    if (arg4 != null) {
        doSomethingElse();
    }
}
```

理解 null 最好的方式是将它当做对象的占位符（placeholder）。这个规则在所有的主流编程规范中都没有提及，但对于全局可维护性来说至关重要。

关于 null 的陷阱会在第 8 章有更进一步的讨论。

1.7.4 undefined

undefined 是一个特殊值，我们常常将它和 null 搞混。其中一个让人颇感困惑之处在于 `null == undefined` 结果是 `true`。然而，这两个值的用途却各不相同。那些没有被初始化的变量都有一个初始值，即 `undefined`，表示这个变量等待被赋值。比如：

```
// 不好的写法
var person;
console.log(person === undefined); //true
```

尽管这段代码能正常工作，但我建议避免在代码中使用 `undefined`。这个值常常和返回“`undefined`”的 `typeof` 运算符混淆。事实上，`typeof` 的行为也很让人费解，因为不管是值是 `undefined` 的变量还是未声明的变量，`typeof` 运算结果都是“`undefined`”。比如：

```
//foo 未被声明
var person;
console.log(typeof person);           //"undefined"
console.log(typeof foo);             //"undefined"
```

在这段代码中，`person` 和 `foo` 都会导致 `typeof` 返回“`undefined`”，哪怕 `person` 和 `foo` 在其他场景中的行为有天壤之别（在语句中使用 `foo` 会报错，而使用 `person` 则不会报错）。

通过禁止使用特殊值 `undefined`，可以有效地确保只在一种情况下 `typeof` 才会返回“`undefined`”：当变量未声明时。如果你使用了一个可能（或可能不会）赋值为一个对象的变量时，则将其赋值为 `null`。

```
// 好的做法
var person = null;
console.log(person === null); //true
```

将变量初始值赋值为 `null` 表明了这个变量的意图，它最终很可能赋值为对象。`typeof` 运算符运算 `null` 的类型时返回“`object`”，这样就可以和 `undefined` 区分开了。

1.7.5 对象直接量

创建对象最流行的一种做法是使用对象直接量，在直接量中直接写出所有属性，这种方式可以取代先显式地创建 `Object` 的实例然后添加属性的这种做法。比如，我们很少见到下面这种写法。

```
// 不好的写法
var book = new Object();
book.title = "Maintainable JavaScript";
book.author = "Nicholas C. Zakas";
```

对象直接量允许你将所有的属性都括在一对花括号内。直接量可以高效地完成非直

接量写法相同的任务，非直接量写法语法看起来更复杂。

当定义对象直接量时，常常在第一行包含左花括号，每一个属性的名值对都独占一行，并保持一个缩进，最后右花括号也独占一行。比如：

```
// 好的写法
var book = {
    title: "Maintainable JavaScript",
    author: "Nicholas C. Zakas"
};
```

这种写法在开源 JavaScript 代码中能经常看到。尽管没有归纳入文档中，Google 的 JavaScript 风格指南非常推荐使用这种写法。Crockford 的编程规范也推荐使用直接量代替 Object 构造函数，但并没有给出具体的书写格式。

1.7.6 数组直接量

和对象直接量类似，数组直接量是 JavaScript 中定义数组最简洁的一种方式。不赞成显式地使用 Array 构造函数来创建数组，比如：

```
// 不好的写法
var colors = new Array("red", "green", "blue");
var numbers = new Array(1, 2, 3, 4);
```

可以使用两个方括号将数组初始元素括起来，来替代使用 Array 构造函数的方式来创建数组。

```
// 好的做法
var colors = [ "red", "green", "blue" ];
var numbers = [ 1, 2, 3, 4 ];
```

在 JavaScript 中，这种模式非常常见。Google 的 JavaScript 风格指南和 Crockford 的编程规范都推荐这样做。

注释

注释是代码中最常见的组成部分。它们是另一种形式的文档，也是程序员最后才舍得花时间去写的。但是，对于代码的总体可维护性而言，注释是非常重要的一环。打开一个没有任何注释的文件就好像趣味冒险，但如果给你的时间有限，这项任务就变成了折磨。适度的添加注释可以解释说明代码的来龙去脉，其他开发者就可以不用从头开始读代码，而是直接去读代码的任意部分。编程风格通常不会包含对注释的风格约定，但我认为从注释的作用即可看出它们的重要性不容忽视。

JavaScript 支持两种不同类型的注释：单行注释和多行注释。

2.1 单行注释

单行注释以两个斜线开始，以行尾结束。

```
// 这是一句单行注释
```

很多人喜欢在双斜线后敲入一个空格，用来让注释文本有一定的偏移。单行注释有三种使用方法。

- 独占一行的注释，用来解释下一行代码。这行注释之前总是有一个空行，且缩进层级和下一行代码保持一致。
- 在代码行的尾部的注释。代码结束到注释之间至少有一个缩进。注释（包括之

前的代码部分) 不应当超过单行最大字符数限制, 如果超过了, 就将这条注释放置于当前代码行的上方。

- 被注释掉的大段代码 (很多编辑器都可以批量注释掉多行代码)。

单行注释不应当以连续多行注释的形式出现, 除非你注释掉一大段代码。只有当需要注释一段很长的文本时才使用多行注释。

这里有一些示例代码。

```
// 好的写法
if (condition) {

    // 如果代码执行到这里, 则表明通过了所有安全性检查
    allowed();
}

// 不好的写法: 注释之前没有空行
if (condition) {
    // 如果代码执行到这里, 则表明通过了所有安全性检查
    allowed();
}

// 不好的写法: 错误的缩进
if (condition) {

    // 如果代码执行到这里, 则表明通过了所有安全性检查
    allowed();
}

// 好的写法
var result = something + somethingElse; // somethingElse 不应当取值为 null

// 不好的写法: 代码和注释之间没有间隔
var result = something + somethingElse; // somethingElse 不应当取值为 null

// 好的写法
// if (condition) {
//     doSomething();
//     thenDoSomethingElse();
// }

// 不好的写法: 这里应当用多行注释
// 接下来的这段代码非常难, 那么, 让我详细解释一下
// 这段代码的作用是首先判断条件是否为真
// 只有为真时才会执行。这里的条件是通过
```

```
// 多个函数计算出来的，在整个会话生命周期内
// 这个值是可以被修改的
if (condition) {
    // 如果代码执行到这里，则表明通过了所有安全性检查
    allowed();
}
```

2.2 多行注释

多行注释可以包裹跨行文本。它以/*开始，以*/结束。多行注释不仅仅可以用来包裹跨行文本，这取决于你。下面这些都是合法的注释。

```
/* 我的注释 */
/* 另一段注释
这段注释包含两行 */
/*
又是一段注释
这段注释同样包含两行
*/
```

尽管从技术的角度看，这些注释都是合法的，但我比较青睐 Java 风格的多行注释。Java 风格的注释至少包含三行：第一行是/*，第二行是以*开始且和上一行的*保持左对齐，最后一行是*/。这种注释看起来像下面这样。

```
/*
 * 另一段注释
 * 这段注释包含两行文本
*/
```

通过在注释块左侧注上星号，会让注释更加清晰。有一些 IDE（比如 NetBean 和 Eclipse）会为你自动插入这些星号。

多行注释总是会出现在将要描述的代码段之前，注释和代码之间没有空行间隔。和单行注释一样，多行注释之前应当有一个空行，且缩进层级和其描述的代码保持一致。来看下面这段例子。

```
// 好的写法
if (condition) {

/*
 * 如果代码执行到这里
 * 说明通过了所有的安全性检测
```

```

        */
    allowed();
}

// 不好的写法：注释之前无空行
if (condition) {
    /*
     * 如果代码执行到这里
     * 说明通过了所有的安全性检测
     */
    allowed();
}

// 不好的写法：星号后没有空格
if (condition) {
    /*
     *如果代码执行到这里
     *说明通过了所有的安全性检测
     */
    allowed();
}

// 不好的写法：错误的缩进
if (condition) {

    /*
     * 如果代码执行到这里
     * 说明通过了所有的安全性检测
     */
    allowed();
}

// 不好的写法：代码尾部注释不要用多行注释格式
var result = something + somethingElse; /*somethingElse 不应当取值为 null*/

```

2.3 使用注释

何时添加注释是程序员经常争论的一个话题。一种通行的指导原则是，当代码不够清晰时添加注释，而当代码很明了时不应当添加注释。比如这个例子中，注释是画蛇添足。

```

// 不好的写法

// 初始化 count
var count = 10;

```

因为代码中初始化 count 的操作是显而易见的。注释并没有提供其他有价值的信息。换个角度讲，如果这个值 10 具有一些特殊的含义，而且无法直接从代码中看出来，这时就有必要添加注释了。

```
// 好的写法  
  
// 改变这个值可能会让它变成青蛙  
var count = 10;
```

当然不可能因为修改了 count 的值它就变成了青蛙，但这的确是一个好的注释写法的例子，因为注释中给出了必要的信息，如果没有注释，你不可能获得这些信息。想象一下如果你修改了 count 的值它真的变成了青蛙，实在是让人困惑不解，一切都源于你没有写这句注释。

因此，添加注释的一般原则是，在需要让代码变得更清晰时添加注释。

2.3.1 难于理解的代码

难于理解的代码通常都应当加注释。根据代码的用途，你可以用单行注释、多行注释，或是混用这两种注释。关键是让其他人更容易读懂这段代码。比如，这段示例代码摘自 YUI 类库中的 Y.mix()方法。

```
// 好的写法  
  
if (mode) {  
  
    /*  
     * 当 mode 为 2 时（原型到原型，对象到对象），这里只递归执行一次  
     * 用来执行原型到原型的合并操作。对象到对象的合并操作  
     * 将会被挂起，在合适的时机执行  
     */  
    if (mode === 2) {  
        Y.mix(receiver.prototype, supplier.prototype, overwrite,  
              whitelist, 0, merge);  
    }  
  
    /*  
     * 根据指定的模式类型，我们可能会从源对象拷贝至原型中，  
     * 或是从原型拷贝至接收对象中  
     */  
    from = mode === 1 || mode === 3 ? supplier.prototype : supplier;  
    to   = mode === 1 || mode === 4 ? receiver.prototype : receiver;
```

```

/*
 * 如果 supplier 或 receiver 不含有原型属性时,
 * 则逻辑结束, 并返回 undefined。如果有原型属性,
 * 则逻辑结束并返回 receiver
 */
if (!from || !to) {
    return receiver;
}
else {
    from = supplier;
    to   = receiver;
}

```

Y.mix()方法使用常量来决定如何处理。mode 参数就表示这些常量其中之一，但仅通过这些数值无法理解它们各自代表的含义。这里的注释非常棒，因为它及时地解释了这里复杂的决策逻辑。

2.3.2 可能被误认为错误的代码

另一个适合添加注释的好时机是当代码看上去有错误时。在团队开发中，总是会有一些好心的开发者在编辑代码时发现他人的代码错误，就立即将它修复。有时这段代码并不是错误的源头，所以“修复”这个错误往往会造成其他错误，因此本次修改应当是可追踪的。当你写的代码有可能会被别的开发者认为有错误时，则需要添加注释。这里是另一段来自 YUI 源码的例子。

```

while (element &&(element = element[axis])) { // 提示：赋值操作
    if ( (all || element[TAG_NAME]) &&
        (!fn || fn(element)) ) {
        return element;
    }
}

```

在这个例子中，开发者在 while 循环控制条件中使用了一个赋值运算符。这不是一种标准用法，并常常被检测工具认为是有问题的。如果你对这段代码不熟悉，读到这段没有注释的代码时，很可能误以为这是一个错误，猜想作者的本意是使用比较运算符==而不是赋值运算符=。这行末尾的注释说明作者是有意为之，即赋值而非比较。这样，其他开发者在读到这段代码时就不会将它“修复”。

2.3.3 浏览器特性 hack

JavaScript 程序员常常会编写一些低效的、不雅的、彻头彻尾的肮脏代码，用来让

低级浏览器正常工作。实际上这种情形是一种特殊的“可能被误认为错误的代码”：这种不明显的做浏览器特性 Hack 的代码可能隐含一些错误。这里有个例子，是摘自 YUI 类库的 Y.DOM.contains()方法。

```
var ret = false;

if ( !needle || !element || !needle[NODE_TYPE] || !element[NODE_TYPE]) {
    ret = false;
} else if (element[CONTAINS]) {
    // 如果 needle 不是 ELEMENT_NODE 时, IE 和 Safari 下会有错误
    if (Y.UA.opera || needle[NODE_TYPE] === 1) {
        ret = element[CONTAINS](needle);
    } else {
        ret = Y_DOM._bruteContains(element, needle);
    }
} else if (element[COMPARE_DOCUMENT_POSITION]) { // gecko
    if (element === needle
|| !(element[COMPARE_DOCUMENT_POSITION](needle) & 16)) {
        ret = true;
    }
}

return ret;
```

这段代码的第 6 行包含一条重要的注释。尽管 IE 和 Safari 中都有内置方法 contains()，但如果 needle 不是一个元素时，这个方法会报错。所以只有当浏览器是 Opera 时才能用这个方法，其他浏览器中 needle 必须是一个元素（nodeType 是 1）。这里关于浏览器的说明同样解释了为什么需要一个 if 语句，这个注释不仅确保将来不会被其他人误改动，而且在代码编写者回过头阅读自己的这段代码时，也会适时地针对新版本的 IE 和 Safari 的兼容情况做出调整。

2.4 文档注释

从技术的角度讲，文档注释并不是 JavaScript 的组成部分，但它们是一种普遍的实践。文档注释有很多种格式，但最流行的一种格式来自于 JavaDoc 文档格式：多行注释以单斜线加双星号（/**）开始，接下来是描述信息，其中使用@符号来表示一个或多个属性。来看一段来自 YUI 的源码的例子。

```
/**
```

返回一个对象，这个对象包含被提供对象的所有属性。

后一个对象的属性会覆盖前一个对象的属性。

传入一个单独的对象，会创建一个它的浅拷贝（shallow copy）。

```
如果需要深拷贝 (deep copy)，请使用'clone()'  
@method merge  
@param {Object} 被合并的一个或多个对象  
@return {Object} 一个新的合并后的对象  
**/  
Y.merge = function () {  
    var args      = arguments,  
        i          = 0,  
        len        = args.length,  
        result     = {};  
  
    for (; i < len; ++i) {  
        Y.mix(result, args[i], true);  
    }  
  
    return result;  
};
```

YUI 类库使用它自己的一个名叫 YUIDoc 的工具来根据这些注释生成文档。但是，它的格式几乎和 JSDoc Toolkit（类库无关的）一模一样，在开源项目中 JSDoc Toolkit 的应用非常广泛，包括 Google 内部的很多开源项目。YUIDoc 和 JSDoc Toolkit 之间的重要区别是，YUIDoc 同时支持文档注释中的 HTML 和 Markdown 格式，而 JSDoc Toolkit 只支持 HTML。

这里强烈推荐你使用文档生成工具来为你的 JavaScript 生成文档。JavaScript 代码注释必须符合你所用的工具支持的格式，但很多文档生成工具都支持 JavaDoc 风格的文档注释。当使用文档注释时，你应当确保对如下内容添加注释。

所有的方法

应当对方法、期望的参数和可能的返回值添加注释描述。

所有的构造函数

应当对自定义类型和期望的参数添加注释描述。

所有包含文档化方法的对象

如果一个对象包含一个或多个附带文档注释的方法，那么这个对象也应当适当地针对文档生成工具添加文档注释。

当然，注释的详细格式和用法最终还是由你所选择的文档生成工具决定的。

第3章

语句和表达式

在 JavaScript 中，诸如 if 和 for 之类的语句有两种写法，使用花括号包裹的多行代码或者不使用花括号的单行代码。比如：

```
// 不好的写法，尽管这是合法的 JavaScript 的代码
if(condition)
    doSomething();

// 不好的写法，尽管是合法的 JavaScript 代码
if(condition) doSomething();

// 好的写法
if (condition) {
    doSomething();
}

// 不好的写法，尽管是合法的 JavaScript 代码
if (condition) { doSomething(); }
```

前两种写法的 if 语句中都没有用花括号，这在 Crockford 的编程规范、jQuery 核心风格指南、Google 的 JavaScript 风格指南以及 Dojo 编程风格指南中都是明确禁止的。默认情况下，省略花括号在 JSLint 和 JSHint 中都会报警告。

绝大多数 JavaScript 程序员都认可这样一点：不论块语句（block statement）包含多行代码还是单行代码，都应当总是使用花括号。因为省略花括号会造成一些困惑。看一下这段代码。

```
if (condition)
    doSomething();
```

```
doSomethingElse();
```

从这段代码中很难看出作者的真正意图。显然这里有错误的，但这个错误到底是缩进错误（最后一行不应当有缩进），还是想执行 if 语句中的第 2 行和第 3 行代码却丢失了花括号？我们不得而知。如果添加了花括号，这个错误就很容易发现。这里另外两个包含错误代码的例子。

```
if (condition) {  
    doSomething();  
}  
    doSomethingElse();  
  
if (condition) {  
    doSomething();  
doSomethingElse();  
}
```

在这两段示例代码中的错误是很明显的，它们都有明显的缩进错误。通过花括号可以很快看出作者的意图，并做出合适的修改，但不会感觉自己修改了最初的代码逻辑。

所有的块语句都应当使用花括号，包括：

- if
- for
- while
- do...while...
- try...catch...finally

3.1 花括号的对齐方式

关于块语句的另一个话题是花括号的对齐方式。有两种主要的花括号对齐风格。第一种风格是，将左花括号放置在块语句中第一句代码的末尾^①，比如：

^① 译注：这里所说的块语句是包含条件（循环）控制语句的，比如这个例子中，块语句的第一句代码实际是 if 语句所在的行。

```
if (condition) {  
    doSomething();  
} else {  
    doSomethingElse();  
}
```

JavaScript 代码的这种风格继承自 Java，这在 Java 编程语言的编程规范中有明确规定。在 Crockford 的编程规范、jQuery 核心风格指南、SproutCore 编程风格指南、Google 的 JavaScript 风格指南以及 Dojo 编程风格指南中都出现过。

花括号的第二种对齐风格是将左花括号放置于块语句首行的下一行。比如：

```
if (condition)  
{  
    doSomething();  
}  
else  
{  
    doSomethingElse();  
}
```

这种风格是随着 C# 流行起来的，因为 Visual Studio 强制使用这种对齐方式。当前并无主流的 JavaScript 编程规范推荐这种风格，Google JavaScript 风格指南明确禁止这种用法，以免导致错误的分号自动插入。我个人也推荐使用第一种花括号对齐格式。

3.2 块语句间隔

块语句首行附近的空白行同样是我们需要考虑的。块语句间隔有三种主要的风格。第一种风格是，在语句名、圆括号和左花括号之间没有空格间隔。

```
if(condition){  
    doSomething();  
}
```

不少程序员喜欢这种风格，因为这种风格看起来很紧凑，而另一些人抱怨说这种紧凑风格实际上破坏了一些易读性。Dojo 编程风格指南推荐使用这种风格。

第二种风格是，在括左圆括号之前和右圆括号之后各添加一个空格。比如：

```
if (condition) {  
    doSomething();  
}
```

有很多程序员青睐这种风格，因为语句类型和条件判断更易读。这种风格是 Crockford 的编程规范和 Google JavaScript 风格指南所推荐的。

第三种风格是在左圆括号后和右圆括号前各添加一个空格，正如下面这段代码所示。

```
if ( condition ) {  
    doSomething();  
}
```

jQuery 核心风格指南文档规定了这种风格，因为它使语句中的各个部分都非常清晰和易读。

我比较倾向于第二种风格，这种风格是第一种和第三种风格的折衷。

3.3 switch 语句

很多程序员对 switch 语句可谓爱恨交加。关于 switch 语句的格式和使用方式也是众说纷纭。其中一些多样性来自于 switch 语句的传承，它源自 C，但在 Java 和 JavaScript 中又没有完全相同的语法。

尽管语法相似，JavaScript 中的 switch 语句的行为和其他语言中是不一样的：switch 语句中可以使用任意类型值，任何表达式都可合法地用于 case 从句。但在其他语言中则必须使用原始值和常量。

3.3.1 缩进

对于 JavaScript 程序员来说，switch 语句的缩进格式是一个有争议的话题。很多人使用 Java 风格的 switch 语句，看起来像下面这样。

```
switch(condition) {  
    case "first":  
        // 代码  
        break;  
  
    case "second":  
        // 代码  
        break;
```

```
case "third":  
    // 代码  
    break;  
  
default:  
    // 代码  
}
```

这种格式的独特之处在于：

- 每条 case 语句相对于 switch 关键字都缩进一个层级。
- 从第二条 case 语句开始，每条 case 语句前后各有一个空行。

几乎所有的风格文档都没有规定何时使用这种格式的 switch 语句，主要是因为这种格式正是很多编辑器自动支持的。

尽管我比较倾向于这种格式，但 Crockford 的编程规范和 Dojo 编程风格指南则提倡另一种格式，这种格式略有不同：

```
switch(condition) {  
    case "first":  
        // 代码  
        break;  
    case "second":  
        // 代码  
        break;  
    case "third":  
        // 代码  
        break;  
    default:  
        // 代码  
}
```

这种写法和前一种写法的主要不同之处在于，case 关键字保持和 switch 关键字左对齐。同样要注意，在语句中并没有空行的存在。JSLint 期望 case 和 switch 具有一致的缩进格式，如果 case 和 switch 缩进不一致时会报警告。可以通过“容忍不规则的间隔”开关选项来开启或关闭 JSLint 的这项检查。如果包含额外的空行，JSLint 是不会报警告的。

其他的编程风格对此没有做规定，这个选择完全是个人偏好问题。

3.3.2 case 语句的“连续执行”

“执行完一个 case 后连续执行（fall through）下一个 case”，这是否是一种广被认可的实践，也是备受争议的一个问题。不小心省略 case 末尾的 break 是很多 bug 的罪魁祸首，因此 Douglas Crockford 提出所有的 case 都应当以 break、return 或 throw 做结尾，但没有给出任何解释。如果某个 case 执行结束后直接进入下一个 case，JSLint 会给出警告。

有很多人认为 case 的连续执行是一种可接受的编程方法，我很同意这种观点，只要程序逻辑非常清晰即可，比如：

```
switch(condition) {  
    // 明显的依次执行  
    case "first":  
    case "second":  
        // 代码  
        break;  
  
    case "third":  
        // 代码  
  
        /* fall through */  
    default:  
        // 代码  
}
```

在这段 switch 语句中，有两个明显的“连续执行”，程序执行完第一个 case 后会继续执行第二个 case，我们认为这种逻辑是合理的（JSLint 也这样认为）。因为第一个 case 语句中没有需要执行的语句，也没有任何其他语句将这两个 case 语句分割开。

第二个例子是 case "third" 执行后继续执行 default 里的逻辑。这个过程已经在注释中明确说明了，这是程序编写者故意为之。在这段代码中，由于加入了注释，我们可以清晰地看出 case 语句何时继续执行、何时终止 switch case，并不会误解为代码错误。当你的代码中出现了 case 语句的连续执行且不包含这句注释（/*fall through */），JSHint 通常会给出警告，加上这句注释告诉 JSHint 这不是一个错误，从而能避免给出警告。

Crockford 的编程规范是禁止 switch 语句中出现连续执行（fall through）的。jQuery

核心风格指南是允许 `case` 的连续执行写法的，Dojo 编程风格指南给出了在连续执行中添加注释的例子。我的建议是，只要是有意为之并且添加了注释，就可以使用 `case` 语句的连续执行。

3.3.3 default

`switch` 语句中另外一个需要讨论的议题是，是否需要 `default`。很多人认为不论何时都不应当省略 `default`，哪怕 `default` 什么也不做。比如：

```
switch(condition) {  
    case "first":  
        // 代码  
        break;  
  
    case "second":  
        // 代码  
        break;  
  
    default:  
        // default 中没有逻辑  
}
```

我们常常在开源 JavaScript 代码中看到这种写法，这段代码包含 `default`，并且补充了一句注释，说明这个 `default` 什么也没做。尽管并没有任何编程规范对此有详细解释，但 Douglas Crockford 的 JavaScript 语言编程规范和 Dojo 编程风格指南将 `default` 作为它们标准的 `switch` 语句格式的组成部分。

我更倾向于在没有默认行为且写了注释的情况下省略 `default`，来看这段例子。

```
switch(condition) {  
    case "first":  
        // 代码  
        break;  
  
    case "second":  
        // 代码  
        break;  
  
    // 没有 default  
}
```

通过这段代码可以看出，代码编写者的意图非常明确，这里的 `switch` 逻辑不应当有默认行为，省掉了不必要的语法结构，也就节省了一些字节。

3.4 with 语句

with 语句可以更改包含的上下文解析变量的方式。通过 with 可以用局部变量和函数的形式来访问特定对象的属性和方法，这样就可以将对象前缀统统省略掉。如果一段代码中书写了很多对象成员，则可以使用 with 语句来缩短这段代码。比如：

```
var book = {  
    title: "Mastheadable JavaScript",  
    author: "Nicholas C. Zakas"  
};  
  
var message = "The book is ";  
  
with (book) {  
    message += title;  
    message += " by " + author;  
}  
}
```

在这个例子中，with 语句花括号内的代码中的 book 的属性都是通过局部变量来读取的，以增强标识符的解析。问题是很难分辨出 title 和 author 出现在哪个位置，也很难分辨出 message 到底是局部变量还是 book 的一个属性。实际上这种困惑对开发者的影响更甚，JavaScript 引擎和压缩工具无法对这段代码进行优化，因为它们无法猜出代码的正确含义。

在严格模式中，with 语句是被明确禁止的，如果使用则报语法错误。这表明 ECMAScript 委员会确信 with 不应当继续使用。Crockford 的编程规范和 Google 的 JavaScript 风格指南禁止使用 with。我也强烈推荐避免使用 with 语句，因为你无法将你的代码运行于严格模式之中（我推荐的一种做法）。

3.5 for 循环

for 循环有两种：一种是传统的 for 循环，是 JavaScript 从 C 和 Java 中继承而来；另外一种是 for-in 循环，用来遍历对象的属性。这两种循环乍一看很类似，但却有着完全不同的用法。传统的 for 循环往往用于遍历数组成员，比如：

```
var values = [ 1, 2, 3, 4, 5, 6, 7 ],  
    i, len;
```

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

有两种方法可以更改循环的执行过程（除了使用 `return` 或 `throw` 语句）。第一种方法是使用 `break` 语句。不管所有的循环迭代有没有执行完毕，使用 `break` 总是可以立即退出循环。比如：

```
var values = [ 1, 2, 3, 4, 5, 6, 7 ],  
    i, len;  
  
for (i=0, len=values.length; i < len; i++) {  
    if (i == 2) {  
        break; // 迭代不会继续  
    }  
    process(values[i]);  
}
```

这里的循环体将执行两次，然后在第三次执行 `process()` 之前就终止循环了，尽管 `values` 数组中的元素超过三个。

第二种更改循环执行过程的方法是使用 `continue`。`continue` 语句可以立即退出（本次）循环，而进入下一次循环迭代，来看下面这个例子。

```
var values = [ 1, 2, 3, 4, 5, 6, 7 ],  
    i, len;  
  
for (i=0, len=values.length; i < len; i++) {  
    if (i == 2) {  
        continue; // 跳过本次迭代  
    }  
    process(values[i]);  
}
```

这里的循环体将执行两次，跳过第三次迭代，而后进入第四次迭代，循环继续执行直到最后一次迭代（如果中途不再会被打断的话）。

Crockford 的编程规范不允许使用 `continue`。他主张代码中与其使用 `continue` 不如使用条件语句。比如，上一个例子可以修改成这样。

```
var values = [ 1, 2, 3, 4, 5, 6, 7 ],  
    i, len;
```

```
for (i=0, len=values.length; i < len; i++) {
    if (i != 2) {
        process(values[i]);
    }
}
```

Crockford 解释说这种方法对于开发者来说更易理解而且不容易出错。Dojo 编程风格指南明确指出可以使用 `continue` 和 `break`。我推荐尽可能避免使用 `continue`，但也没有理由完全禁止使用，它的使用应当根据代码可读性来决定。

当使用了 `continue` 时，JSLint 会给出警告。而 JSHint 不会给出警告。

3.6 for-in 循环

for-in 循环是用来遍历对象属性的。不用定义任何控制条件，循环将会有条不紊地遍历每个对象属性，并返回属性名而不是值。比如：

```
var prop;

for (prop in object) {
    console.log("Property name is " + prop);
    console.log("Property value is " + object[prop]);
}
```

for-in 循环有一个问题，就是它不仅遍历对象的实例属性（instance property），同样还遍历从原型继承来的属性。当遍历自定义对象的属性时，往往会因为意外的结果而终止。出于这个原因，最好使用 `hasOwnProperty()` 方法来为 for-in 循环过滤出实例属性。来看下面这个例子。

```
var prop;

for (prop in object) {
    if (object.hasOwnProperty(prop)) {
        console.log("Property name is " + prop);
        console.log("Property value is " + object[prop]);
    }
}
```

Crockford 的编程规范要求所有的 for-in 循环都必须使用 `hasOwnProperty()`。默认情况下，对于循环体中没有使用 `hasOwnProperty()` 的 for-in 循环，JSLint 和 JSHint 都会给出警告（可以手动关掉 JSLint 和 JSHint 的这项检查）。我推荐总是在 for-in 循

环中使用 `hasOwnProperty()`，除非你想查找原型链，这时就应当补充注释，比如：

```
var prop;

for (prop in object) { // 包含对原型链的遍历
    console.log("Property name is " + prop);
    console.log("Property value is " + object[prop]);
}
```

关于 `for-in` 循环，还有一点需要注意，即 `for-in` 循环是用来遍历对象的。一个常见的错误用法是使用 `for-in` 循环来遍历数组成员，正如下面这段代码显示的那样。

```
// 不好的用法
var values = [ 1, 2, 3, 4, 5, 6, 7],
    i;

for (i in values) {
    process(items[i]);
}
```

这种用法在 Crockford 的编程规范、Google 的 JavaScript 风格指南中是禁止的，因为这会造成潜在的错误。记住，`for-in` 循环是用来对实例对象和原型链中的键（key）做遍历的，而不是用来遍历包含数字索引的数组的。因此 `for-in` 循环不应当用于这种场景。

变量、函数和运算符

JavaScript 编程的本质是编写一个个的函数来完成任务。在函数内部，变量和运算符可以通过移动操作字节来使某件事发生。因此在讨论过基本的 JavaScript 书写格式化之后，接下来关注如何使用函数、变量和运算符来减少复杂度和增强可读性就显得十分重要了。

4.1 变量声明

变量声明是通过 var 语句来完成的。JavaScript 中允许多次使用 var 语句，此外 var 语句几乎可以用在 JavaScript 脚本中的任意地方。这对开发者来说是非常有意思的一件事，因为不论 var 语句是否真正会被执行，所有的 var 语句都提前到包含这段逻辑的函数的顶部执行。^① 比如：

```
function doSomething() {  
    var result = 10 + value;  
    var value = 10;  
    return result;  
}
```

在这段代码中，变量 value 在声明之前就参与了运算，这是完全合法的，尽管这会造成 result 的计算结果是一个特殊值 NaN。要想了解其中的原理，你需要清楚这段代码被 JavaScript 理解为如下模样。

^① 译注：《JavaScript 权威指南》（第六版）的 3.10.1 小节详细讲解了变量的声明提前。

```
function doSomething() {  
    var result;  
    var value;  
  
    result = 10 + value;  
    value = 10;  
  
    return result;  
}
```

两个 `var` 语句提前至函数的顶部；初始化逻辑紧跟其后。在第 6 行用到变量 `value` 的时候它的值是一个特殊值 `undefined`，因此 `result` 的值就是 `Nan`（非数字）。在那之后 `value` 才最终赋值为 10。

在某些场景中，开发者往往会漏掉变量的声明提前，`for` 语句就是其中一个常见的场景，在 `for` 语句中的变量被声明为初始化逻辑的一部分。

```
function doSomethingWithItems(items) {  
  
    for (var i=0, len=items.length; i < len; i++) {  
        doSomething(items[i]);  
    }  
}
```

在 ECMAScript5 之前，JavaScript 中并没有块级变量声明（block-level variable declaration），因此这段代码实际上等价于下面这段代码。

```
function doSomethingWithItems(items) {  
  
    var i, len;  
  
    for (i=0, len=items.length; i < len; i++) {  
        doSomething(items[i]);  
    }  
}
```

变量声明提前意味着：在函数内部任意地方定义变量和在函数顶部定义变量是完全一样的。因此，一种流行的风格是将所有变量声明放在函数顶部而不是散落在各个角落。简言之，依照这种风格写出的代码逻辑和 JavaScript 引擎解析这段代码的习惯是非常相似的。

我建议你总是将局部变量的定义作为函数内第一条语句。Crockford 的编程规范、SproutCore 编程风格指南和 Dojo 编程风格指南也推荐这样做。

```
function doSomethingWithItems(items) {  
  
    var i, len;  
    var value = 10;  
    var result = value + 10;  
  
    for (i=0, len=items.length; i < len; i++) {  
        doSomething(items[i]);  
    }  
}
```

Crockford 还进一步推荐在函数顶部使用单 var 语句^①。

```
function doSomethingWithItems(items) {  
  
    var i, len,  
        value = 10,  
        result = value + 10;  
  
    for (i=0, len=items.length; i < len; i++) {  
        doSomething(items[i]);  
    }  
}
```

Dojo 编程风格指南规定，只有当变量之间有关联性时，才允许使用单 var 语句。

我个人比较倾向于将所有的 var 语句合并为一个语句，每个变量的初始化独占一行。赋值运算符应当对齐。对于那些没有初始值的变量来说，它们应当出现在 var 语句的尾部，比如下面这段代码。

```
function doSomethingWithItems(items) {  
  
    var value    = 10,  
        result   = value + 10,  
        i,  
        len;  
  
    for (i=0, len=items.length; i < len; i++) {
```

^① 译注：在《JavaScript Patterns》(O'Reilly 出版 2010 年出版) 中也推荐使用单 var 语句，并将其定义为一种最佳实践，《JavaScript Pattern》中作者细致充分地讲解了声明提前和单 var 模式。

```
        doSomething(items[i]);
    }
}
```

为了保持成本最低，我推荐合并 var 语句，这可以让你的代码更短、下载更快。

4.2 函数声明

和变量声明一样，函数声明也会被 JavaScript 引擎提前。因此，在代码中函数的调用可以出现在函数声明之前。

```
// 不好的写法
doSomething();

function doSomething() {
    alert("Hello world!");
}
```

这段代码是可以正常运行的，因为 JavaScript 引擎将这段代码解析为：

```
function doSomething() {
    alert("Hello world!");
}

doSomething();
```

由于 JavaScript 的这种行为，我们推荐总是先声明 JavaScript 函数然后使用函数。Crockford 的编程规范里包含这种设计。Crockford 的编程规范还推荐函数内部的局部函数应当紧接着变量声明之后声明，比如：

```
function doSomethingWithItems(items) {

    var i, len,
        value = 10,
        result = value + 10;

    function doSomething(item) {
        // 代码逻辑
    }

    for (i=0, len=items.length; i < len; i++) {
        doSomething(items[i]);
    }
}
```

```
    }
}
```

当函数在声明之前就使用时，在 JSLint 和 JSHint 中都会给出警告。

此外，函数声明不应当出现在语句块之内。比如，这段代码就不会按照我们的意图来执行。

```
// 不好的写法
if (condition) {
    function doSomething() {
        alert("Hi!");
    }
} else {
    function doSomething() {
        alert("Yo!");
    }
}
```

这段代码在不同浏览器中的运行结果也是不尽相同的。不管 `condition` 的计算结果如何，大多数浏览器都会自动使用第二个声明。而 Firefox 则根据 `condition` 的计算结果选用合适的函数声明。这种场景是 ECMAScript 的一个灰色地带，应当尽可能地避免。函数声明应当在条件语句的外部使用。这种模式也是 Google 的 JavaScript 风格指南明确禁止的。

4.3 函数调用间隔

一般情况下，对于函数调用写法推荐的风格是，在函数名和左括号之间没有空格。这样做是为了将它和块语句（block statement）区分开来。比如：

```
// 好的写法
doSomething(item);

// 不好的写法：看起来像一个块语句
doSomething (item);

// 用来做对比的块语句
while (item) {
    // 代码逻辑
}
```

Crockford 的编程规范对此有明确的规定。Google 的 JavaScript 风格指南、SproutCore 编程风格指南以及 Dojo 编程风格指南对比没有明确规定，但在它们的示例代码中使用了这种风格。

jQuery 核心风格指南中的规定更进一步，它规定应当在左括号之后和右括号之前都加上空格。比如：

```
// jQuery 风格
doSomething( item );
```

这种风格是为了让参数更易读。jQuery 核心风格指南同样列出了这种风格的一些例外情况，特别是和这些传入单参数的函数相关的场景，这些参数包括对象直接量、数组直接量、函数表达式或者字符串。

```
// jQuery 例外情况
doSomething(function() {});
doSomething({ item: item });
doSomething([ item ]);
doSomething("Hi!");
```

通常情况下，如果一个风格有超过一个例外，那么这种风格是不好的，因为这会给开发者带来一些困惑。

4.4 立即调用的函数

JavaScript 中允许声明匿名函数（本身没有命名的函数），并将匿名函数赋值给变量或者属性。

```
var doSomething = function() {
    // 函数体
};
```

这种匿名函数同样可以通过在最后加上一对圆括号来立即执行并返回一个值，然后将这个值赋值给变量。

```
// 不好的写法
var value = function() {
    // 函数体
};
```

```
    return {
      message: "Hi"
    }
}();
```

在这个例子中，`value` 最终被赋值为一个对象，因为函数立即执行了。这种模式的问题在于，会让人误以为将一个匿名函数赋值给了这个变量。除非读完整段代码看到最后一行的那对圆括号，否则你不会知道是将函数赋值给变量还是将函数的执行结果赋值给变量。这种困惑会影响代码的可读性。

为了让立即执行的函数能够被一眼看出来，可以将函数用一对圆括号包裹起来，比如：

```
// 好的写法
var value = (function() {

  // 函数体

  return {
    message: "Hi"
  }
})();
```

这段代码在第一行就有了一个标识符（左圆括号），表明这是一个立即执行的函数。添加一对圆括号并不会改变代码的逻辑。Crockford 的编程规范推荐这种模式，在省略圆括号的情况下 JSLint 会报警告。

4.5 严格模式

ECMAScript5 引入了“严格模式”（strict mode），希望通过这种方式来谨慎地解析执行 JavaScript，以减少错误。通过使用如下指令脚本以严格模式执行。

```
"use strict";
```

尽管这看起来像是一个没有赋值给变量的字符串，但 ECMAScript5 JavaScript 引擎还是会将其识别为一条指令，以严格模式来解析代码。这条编译指令（pragma）不仅用于全局，也适用于局部，比如一个函数内。但是不推荐将“`use strict`”用在全

局作用域中（尽管所有流行的编程规范中都没有提及），因为这会让文件中的所有代码都以严格模式来解析，所以如果你将 11 个文件连接合并成一个文件时，当其中一个文件在全局作用域中启用了严格模式，则所有的代码都将以严格模式解析。由于严格模式中的运算符规则和在非严格模式下的情形有很大不同，因此其他文件中的（非严格模式下的）代码很可能会报错。因此，最好不要在全局作用域中使用“use strict”，来看一些例子。

```
// 不好的写法 - 全局的严格模式
"use strict";
function doSomething() {
    // 代码
}

// 好的写法
function doSomething() {
    "use strict";
    // 代码
}
```

如果你希望在多个文件中应用严格模式而不必写很多行“use strict”的话，可以使用立即执行的函数。

```
// 好的写法
(function() {
    "use strict";
    function doSomething() {
        // 代码
    }
    function doSomethingElse() {
        // 代码
    }
})();
```

在这段代码中，doSomething()和doSomethingElse()都以严格模式运行，因为它们都包含在一个立即执行的函数中，这个函数里包含“use strict”。

当“use strict”出现在函数体之外，在JSLint和JSHint中都会给出警告。JSLint和JSHint希望所有函数都默认包含“use strict”，当然在这两个工具中都可以关闭这个选项。我推荐尽可能使用严格模式，以减少常见错误的发生。

4.6 相等

由于 JavaScript 具有强制类型转换机制（type coercion），JavaScript 中的判断相等操作是很微妙的。对于某些运算来说，为了得到成功的结果，强制类型转换会驱使某种类型的变量自动转换成其他不同类型，这种情形往往会造成意想不到的结果。

发生强制类型转换最常见的场景就是，使用了判断相等运算符`==`和`!=`的时候。当要比较的两个值的类型不同时，这两个运算符都会有强制类型转换（当数据类型相同时则不会有强制类型转换）。但在很多实际情况中，代码并不按照我所期望的方式运行。

如果将数字和字符串进行比较，字符串会首先转换为数字，然后执行比较，看下面这个例子。

```
// 比较数字 5 和字符串 5
console.log(5 == "5"); // true

// 比较数字 25 和十六进制的字符串 25
console.log(25 == "0x19"); // true
```

当发生强制类型转换时，字符串会被转换为数字，类似使用`Number()`转换函数。因为`Number()`可以正确解析十六进制的格式，它会将看起来像十六进制的数字转换为十进制数，然后再进行比较。

如果一个布尔值和数字比较，布尔值会首先转换为数字，然后进行比较。`false` 值变为 0、`true` 变为 1。来看例子：

```
// 数字 1 和 true
console.log(1 == true); // true

// 数字 0 和 false
console.log(0 == false); // true

// 数字 2 和 true
console.log(2 == true); // false
```

如果其中一个值是对象而另一个不是，则会首先调用对象的`valueOf()`方法，得到原

始类型值再进行比较。如果没有定义 `valueOf()`，则调用 `toString()`。之后的比较操作就和上文提到的多类型值比较的情形一样了，例如：

```
var object = {
    toString: function() {
        return "0x19";
    }
};

console.log(object == 25); // true
```

这个对象被认为与数字 25 相等，因为 `toString()`方法返回了十六进制的字符串“0x19”，这个值先被转换为数字，然后和 25 进行比较。

还有一种涉及到强制类型转换的场景和 `null` 和 `undefined` 有关。根据 ECMAScript 标准规范的描述，这两个特殊值被认为是相等的。

```
console.log(null == undefined); //true
```

由于强制类型转换的缘故，我们推荐不要使用`==`和`!=`，而是应当使用`===`和`!==`。用这些运算符作比较不会涉及强制类型转换。因此，如果两个值的类型不一样，则认为它们不相等，这样就可以让你的比较语句执行比较时行为一致。来看一下下面几个例子，观察一下`==`和`===`之间的区别。

```
// 数字 5 和字符串 5
console.log(5 == "5"); // true
console.log(5 === "5"); // false

// 数字 25 和十六进制字符串 25
console.log(25 == "0x19"); // true
console.log(25 === "0x19"); // false

// 数字 1 和 true
console.log(1 == true); // true
console.log(1 === true); // false

// 数字 0 和 false
console.log(0 == false); // true
console.log(0 === false); // false

// 数字 2 和 true
console.log(2 == true); // false
console.log(2 === true); // false
```

```

var object = {
    toString: function() {
        return "0x19";
    }
};

// 一个对象和 25
console.log(object == 25); // true
console.log(object === 25); // false

// Null 和 undefined
console.log(null == undefined); // true
console.log(null === undefined); // false

```

Crockford 的编程规范、jQuery 核心风格指南、SproutCore 编程风格指南推荐使用`==`和`!=`。Crockford 编程规范强调所有情形都应当使用`==`和`!=`，尤其是涉及到假值比较的场景（那些强制转换为布尔值后是`false`的值，比如`0`、空字符串、`null`、`undefined`）。jQuery 核心风格指南则允许在和`null`比较时用`==`，因为这时程序编写者往往是想判断值是否为`null`或`undefined`。我推荐毫无例外地总是使用`==`和`!=`。

如果使用了`==`或`!=`，JSLint 默认会报警告，而 JSHint 则会针对使用`==`或`!=`来比较假值的情形报警告。你可以通过添加`eqlreq`选项来启用针对所有`==`和`!=`的警告^①。

4.6.1 eval()

在 JavaScript 中，`eval()`的参数是一个字符串，`eval()`会将传入的字符串当作代码来执行。开发者可以通过这个函数来载入外部的 JavaScript 代码，或者随即生成 JavaScript 代码并执行它。比如：

```

eval("alert('Hi!')");

var count = 10;
var number = eval("5 + count");
console.log(count); // 15

```

在 JavaScript 中 `eval()`并不是唯一可以执行 JavaScript 字符串的函数，使用 `Function` 构造函数亦可以做到这一点，`setTimeout()`和`setInterval()`也可以。来看一些例子。

```
var myfunc = new Function("alert('Hi!')");
```

^① 译注：在 js 文件的开头使用注释开启，比如`/*jshint eqeqeq:true,undef:false*/`。

```
setTimeout("document.body.style.background='red'", 50);

setInterval("document.title = 'It is now '" + (new Date()), 1000);
```

在大多数 JavaScript 社区之中，人们都认为这些代码是糟糕的实践。尽管在 JavaScript 类库中 eval()，可能会经常用到（通常和 JSON 操作有关），另外三种用法即使有也十分罕见。一个通用的原则是，严禁使用 Function，并且只在别无他法时使用 eval()。setTimeout()和 setInterval()也是可以使用的，但不要用字符串形式而要用函数。

```
setTimeout(function() {
    document.body.style.background='red';
}, 50);

setInterval(function() {
    document.title = 'It is now ' + (new Date());
}, 1000);
```

Crockford 的编程规范禁止使用 eval() 和 Function，也禁止给 setTimeout() 和 setInterval() 传入字符串参数。jQuery 核心风格指南禁止使用 eval()，但有一个唯一的例外，即涉及到回调中解析 JSON 的情形。Google 的 JavaScript 风格指南只允许在将 Ajax 的返回值转换为 JavaScript 值的场景下使用 eval()。

默认情况下，对于使用 eval()、Function、setTimeout() 和 setInterval() 的情形，JSLint 和 JSHint 都会给出警告。

ECMAScript 5 严格模式对于 eval() 有着严格的限制，禁止在一个封闭的作用域中使用它创建新变量或者函数。这条限制帮助我们避免了 eval() 先天的安全漏洞^①。然而，如果实在没有别的方法来完成当前任务，这时依然推荐使用 eval()。

4.6.2 原始包装类型

JavaScript 中的一个不易被了解且被常常误解的方面是，这门语言对原始包装类型（primitive wrapper types）的依赖。JavaScript 里有 3 种原始包装类型：String、Boolean 和 Number。每种类型都代表全局作用域中的一个构造函数，并分别表示各自对

^① 译注：严格模式下，传入 eval() 的字符串无法在调用函数所在的上下文声明变量或函数，非严格模式下是可以这样做的，更多详情请参照《JavaScript 权威指南（第六版）》5.7.3 小节。

应的原始值的对象。原始包装类型的主要作用是让原始值具有对象般的行为^①，比如：

```
var name = "Nicholas";
console.log(name.toUpperCase());
```

尽管 `name` 是一个字符串，是原始类型不是对象，但你仍然可以使用诸如 `toUpperCase()` 之类的方法，即将字符串当作对象来对待。这种做法之所以行得通，是因为在这条语句的表象背后 JavaScript 引擎创建了 `String` 类型的新实例，紧跟着就被销毁了，当再次需要时就会又创建另外一个对象。你可以通过给一个字符串增加属性来检验这种行为。

```
var name = "Nicholas";
name.author = true;
console.log(name.author); // undefined
```

在第 2 行结束后，`author` 属性就不见了。因为表示这个字符串的临时 `String` 对象在第 2 行执行结束后就销毁了，在第 3 行中又创建了一个新 `String` 对象。同样你也可以自己手动创建这些对象。

```
// 不好的做法
var name = new String("Nicholas");
var author = new Boolean(true);
var count = new Number(10);
```

尽管我们可以使用这些包装类型，但我强烈推荐大家避免使用它们。开发者的思路常常会在对象和原始值之间跳来跳去，这样会增加出 `bug` 的概率，从而使开发者陷入困惑。你也没有理由自己手动创建这些对象。

Google 的 JavaScript 风格指南禁止使用原始包装类型。当你使用 `String`、`Number` 或 `Boolean` 来创建新对象时，`JSLint` 和 `JSHint` 都会给出警告。

^① 译注：《JavaScript 权威指南（第六版）》3.6 “包装对象” 详细讲解了包装对象和包装类型的细节，需要尤为注意的一点是，原始值本身不具有对象特性，比如 `1.toString()` 是报错的，必须这样做：`var a = 1; a.toString()`。

第二部分

编程实践

“构建软件设计的方法有两种：一种是把软件做得很简单以至于明显找不到缺陷；另一种是把它做得很复杂以至于找不到明显的缺陷。”

——C.A.R. Hoare，1980年图灵奖获得者^①。

在本书的第一部分里，我们主要讨论 JavaScript 的代码风格规范（style guideline）。代码风格规范的目的是在多人协作的场景下使代码具有一致性。关于如何解决一般性的问题的讨论是不包含在风格规范中的，那是编程实践中的内容。

编程实践是另外一类编程规范。代码风格规范只关心代码的呈现，而编程实践则关心编码的结果。你可以将编程实践看作是“秘方”——它们指引开发者以某种方式编写代码，这样做的结果是已知的。如果你使用过一些设计模式比如 MVC 中的观察者模式，那么你已经对编程实践很熟悉了。设计模式是编程实践的组成部分，专用于解决和软件组织相关的特定问题。

这一部分的编程实践只会涵盖很小的问题。其中一些实践是和设计模式相关的，另外更多的内容只是增强你的代码总体质量的一些简单小技巧。

JSLint 和 JSHint 除了对代码风格进行检查，也包含了一些关于编程实践方面的警告。非常推荐大家在 JavaScript 开发工作中使用这些工具，来确保不会发生那些看上去不起眼但又难于发现的错误。

^① 译注：图灵奖是美国计算机学会（ACM）设立的计算机界的最高奖。

UI 层的松耦合

在 Web 开发中，用户界面（User Interface, UI）是由三个彼此隔离又相互作用的层定义的。

- HTML 用来定义页面的数据和语义。
- CSS 用来给页面添加样式，创建视觉特征。
- JavaScript 用来给页面添加行为，使其更具交互性。

UI 层次关系是这样的，HTML 在最底层，CSS 和 JavaScript 在高层，如图 5-1 所示。

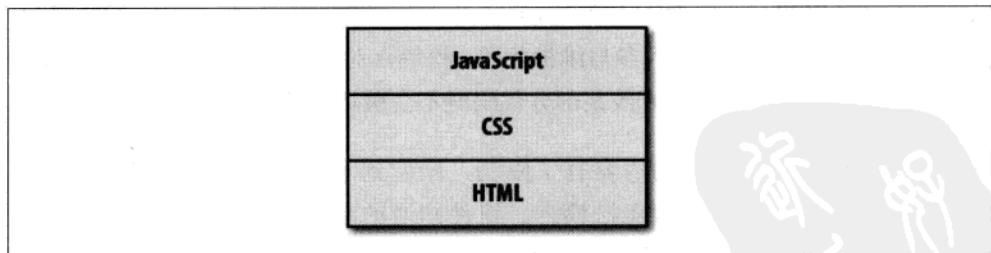


图 5-1 Web UI 的分层

在实际场景中，CSS 和 JavaScript 更像是兄弟关系而非依赖关系（JavaScript 依赖 CSS）。一个页面很可能只有 HTML 和 CSS 而没有 JavaScript，同样，一个页面也可以只有 HTML 和 JavaScript 而没有 CSS。我更愿意以图 5-2 所示的分层来理解三

者的关系。

为了更多地增加分层的合理性和消除依赖性，我们认为在所有 Web UI 中，CSS 和 JavaScript 是同等重要的。比如，JavaScript 的正确运行不应当依赖 CSS——在缺少 CSS 的情况下也要能够正确运行，尽管两者之间可能有些互动。

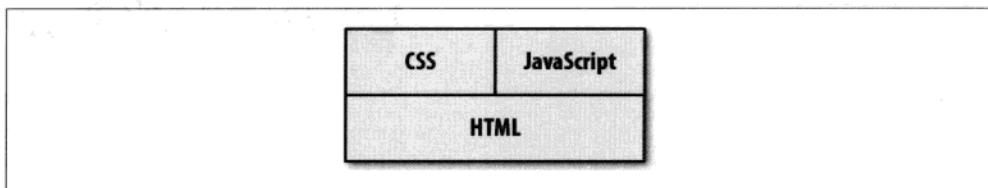


图 5-2 更新后的 Web UI 分层

5.1 什么是松耦合

很多设计模式就是为了解决紧耦合的问题。如果两个组件耦合太紧，则说明一个组件和另一个组件直接相关，这样的话，如果修改一个组件的逻辑，那么另外一个组件的逻辑也需修改。比如，假设有一个名为 `error` 的 CSS 类名，它是贯穿整个站点的，它被嵌入到 HTML 之中。如果有一天你觉得 `error` 的取名并不合适，想将它改为 `warning`，你不仅需要修改 CSS 还要修改用到这个 `className` 的 HTML。HTML 和 CSS 紧耦合在一起。这只是一个简单的例子。想象一下，如果一个系统包含上百个组件，那这简直就是一场噩梦。

当你能够做到修改一个组件而不需要更改其他的组件时，你就做到了松耦合。对于多大型系统来说，有很多人参与维护代码，松耦合对于代码可维护性来说至关重要。你绝对希望开发人员在修改某部分代码时不会破坏其他人的代码。

当一个大系统的每个组件的内容有了限制，就做到了松耦合。本质上讲，每个组件需要保持足够瘦身来确保松耦合。组件知道的越少，就越有利于形成整个系统。

有一点需要注意：在一起工作的组件无法达到“无耦合”(no coupling)。在所有系统中，组件之间总要共享一些信息来完成各自的工作。这很好理解，我们的目标是确保对一个组件的修改不会经常性地影响其他部分。

如果一个 Web UI 是松耦合的，则很容易调试。和文本或结构相关的问题，通过查找 HTML 即可定位。当发生了样式相关的问题，你知道问题出现在 CSS 中。最后，对于那些行为相关的问题，你直接去 JavaScript 中找到问题所在，这种能力是 Web 界面的可维护性的核心部分。

5.2 将 JavaScript 从 CSS 中抽离

在 IE8 和更早版本的浏览器中有一个特性让人爱少恨多，即 CSS 表达式（CSS expression）。CSS 表达式允许你将 JavaScript 直接插入到 CSS 中，这样可以在 CSS 代码中直接执行运算或其他操作。比如，下面这段代码设置了元素宽度以匹配浏览器宽度。

```
/* 不好的写法 */
.box {
    width: expression(document.body.offsetWidth + "px");
}
```

CSS 表达式包裹在一个特殊的 `expression()` 函数中，可以给它传入任意 JavaScript 代码。浏览器会以高频率重复计算 CSS 表达式，这严重影响了性能，甚至 Steve Souder 的那本《高性能网站建设指南》也特意提到这一点，要避免使用 CSS 表达式（规则 7：避免使用 CSS 表达式）。

除了性能问题之外，在 CSS 中嵌入 JavaScript 代码对于代码维护来说亦是一场噩梦。这方面我有着切身的体会。在 2005 年，有一个 JavaScript bug 分配到我这里，从一开始就很棘手。这个 bug 只在 IE 下才有，并且只有当浏览器窗口大小发生几次改变的时候才会出现。在当时 IE 下，最好的 JavaScript 调试器就是 Visual Studio，但我无论如何也没办法找到错误。我花了一整天时间来设置断点和插入 `alert()` 语句试图找到问题的根源。

在快下班的时候，我被逼无奈使用了我最不喜欢的调试方法：逐段删除代码。我一个接一个地删除 JavaScript 代码，尝试复现这个问题。我很快就删除了页面中所有的 JavaScript 代码，但问题依然存在，这让我困惑至极。我简直不敢相信自己的眼睛。页面中没有任何 JavaScript 代码，却报 JavaScript 错误——真的是见鬼了。

直到今天，我依然很纳闷是什么原因促使我去关注 CSS。那时我甚至不知道要去 CSS

中查什么。只是小心翼翼地从头开始看，一点点地向下滚动我的代码，试图找出一些不一样之处。最终，我发现了症结所在：CSS 表达式。当我将其删除，JavaScript 错误也随之不见了。

正是这次经历促使我提炼出了本章所提到的准则。我花了一整天时间在 JavaScript 中查找脚本错误，罪魁祸首竟然是 CSS。解决这个错误并不难，但是在我认为不会出错的代码里找错误，这着实是一件可笑的浪费时间的事。

幸运的是，IE9 不再支持 CSS 表达式了，但是老版本的 IE 依然可以运行 CSS 表达式。尽管我们很少用 CSS 表达式去实现一些罕见的功能，来让低版本浏览器里也达到和高级浏览器一致的表现，但还是要克制住这份冲动，避免不必要的浪费时间和精力。将 JavaScript 从 CSS 中抽离出来。

5.3 将 CSS 从 JavaScript 中抽离

有时候，保持 CSS 和 JavaScript 之间清晰的分离是很有挑战的。这两门语言相互协作得很不错，所以我们经常将样式数据和 JavaScript 混写在一起。通过脚本修改样式最流行的一种方法是，直接修改 DOM 元素的 `style` 属性。`style` 属性是一个对象，包含了可以读取和修改的 CSS 属性。比如，你可以像下面这样修改元素里的文本颜色。

```
// 不好的写法
element.style.color = "red";
```

我们经常看到使用这种方法来修改多个样式属性的代码段，比如：

```
// 不好的写法
element.style.color = "red";
element.style.left = "10px";
element.style.top = "100px";
element.style.visibility = "visible";
```

这种方法是有问题的，因为样式信息是通过 JavaScript 而非 CSS 来承载的。当出现了样式问题，你通常首先会先去查找 CSS。直到你精疲力竭地排除了所有可能性，才会去 JavaScript 中查找样式信息。

开发者修改 `style` 对象还有一种方式，给 `cssText` 属性赋值整个 CSS 字符串，看下

面这个例子。

```
// 不好的写法
element.style.cssText = "color: red; left: 10px; top: 100px; visibility: hidden";
```

使用 `cssText` 属性只是一次性设置多个 CSS 属性的一种快捷写法。这种模式同样有问题，比如在设置单个属性时：将样式信息写入 JavaScript 带来了可维护性问题。

将 CSS 从 JavaScript 中抽离意味着所有的样式信息都应当保持在 CSS 中。当需要通过 JavaScript 来修改元素样式的时候，最佳方法是操作 CSS 的 `className`，比如，我想在页面中显示一个对话框，在 CSS 中的样式定义是像下面这样的。

```
.reveal {
    color: red;
    left: 10px;
    top: 100px;
    visibility: visible;
}
```

然后，在 JavaScript 中像这样将样式添加至元素。

```
// 好的写法 - 原生方法
element.className += " reveal";

// 好的写法 - HTML5
element.classList.add("reveal");

// 好的写法 - YUI
Y.one(element).addClass("reveal");

// 好的写法 - jQuery
$(element).addClass("reveal");

// 好的写法 - Dojo
dojo.addClass(element, "reveal");
```

由于 CSS 的 `className` 可以成为 CSS 和 JavaScript 之间通信的桥梁。在页面的生命周期中，JavaScript 可以随意添加和删除元素的 `className`。而 `className` 所定义的样式则在 CSS 代码之中。任何时刻，CSS 中的样式都是可以修改的，而不必更新 JavaScript。JavaScript 不应当直接操作样式，以便保持和 CSS 的松耦合。



有一种使用 `style` 属性的情形是可以接收的：当你需要给页面中的元素作定位，使其相对于另外一个元素或整个页面重新定位。这种计算是无法在 CSS 中完成的，因此这时是可以使用 `style.top`、`style.left`、`style.bottom` 和 `style.right` 来对元素作正确定位的。在 CSS 中定义这个元素的默认属性，而在 JavaScript 中修改这些默认值。

5.4 将 JavaScript 从 HTML 中抽离

很多人学习 JavaScript 之初所做的第一件事是，将脚本嵌入到 HTML 中来运行。有很多种方式。第一种方式是使用 `on` 属性（比如 `onclick`）来绑定一个事件处理程序。

```
<!-- 不好的写法 -->
<button onclick="doSomething()" id="action-btn">Click Me</button>
```

这种写法在 2000 年的时候非常流行，多数网站都采用了这种写法。HTML 代码中放满了 `onclick` 和其他的事件处理程序，很多元素都包含这样的属性。尽管这种代码在多数场景下是正常工作的，但却是两个 UI 层（HTML 和 JavaScript）的深耦合，因此这种写法是有一些问题的。

首先，当按钮上发生点击事件时，`doSomething()` 函数必须存在。那些在 2000 年左右开发过网站的人对这个问题非常熟悉。包含 `doSomething()` 的代码可能是从外部文件载入或存在于 HTML 文件的后面某处。不管哪种情形，都有可能出现用户点击按钮时这个函数还不存在，这时就会报 JavaScript 错误。页面会弹出错误信息或者点击事件不会有任何响应。两种情形都是我们所不希望发生的。

第二个问题在于可维护性方面。如果你修改了 `doSomething()` 的函数名，将会发生什么事情？如果这时点击按钮调用了别的函数又会怎样呢？在这两个例子中，你需要同时修改 JavaScript 和 HTML 代码。这就是典型的紧耦合的代码。

你的绝大多数（并非所有的）JavaScript 代码都应当包含在外部文件中，并在页面中通过`<script>` 标签来引用。在 HTML 代码中，也不应当直接给 `on` 属性挂载事件处理程序。相反，一旦外部脚本文件加载至页面，则使用 JavaScript 方法来添加事件处理程序。对于支持 2 级 DOM 模型的浏览器来说，用如下这段代码可以完成上

一个例子中的功能。

```
function doSomething() {  
    // 代码  
}  
  
var btn = document.getElementById("action-btn");  
btn.addEventListener("click", doSomething, false);
```

这种方法的优势在于，函数 `doSomething()` 的定义和事件处理程序的绑定都是在一个文件中完成的。如果函数名称需要修改，则只需修改一个文件；如果点击发生时想额外做一些动作，也只需在一处做修改。

IE8 及其更早的版本不支持 `addEventListener()` 函数，因此你需要一个标准的函数将这些差异性做封装。

```
function addListener(target, type, handler) {  
    if (target.addEventListener) {  
        target.addEventListener(type, handler, false);  
    } else if (target.attachEvent) {  
        target.attachEvent("on" + type, handler);  
    } else {  
        target["on" + type] = handler;  
    }  
}
```

这个函数可以做到在各种浏览器中给一个元素添加事件处理程序，甚至可以降级到支持给 0 级 DOM 模型对象的 `on` 属性赋值处理程序（只有在非常古老的浏览器，比如 Netscape 4 中，才会执行这一步，因此这段代码可以在所有情形下都正常工作）。我们常常像下面这样来使用这个方法。

```
function doSomething() {  
    // 代码  
}  
  
var btn = document.getElementById("action-btn");  
addListener(btn, "click", doSomething);
```

如果你用了 JavaScript 类库，可以使用类库提供的方法来给元素挂载事件处理程序。这里给出一些流行的类库中的实例代码。

```
// YUI  
Y.one("#action-btn").on("click", doSomething);
```

```
// jQuery
$("#action-btn").on("click", doSomething);

// Dojo
var btn = dojo.byId("action-btn");
dojo.connect(btn, "click", doSomething);
```

在 HTML 中嵌入 JavaScript 的另一种方法是使用`<script>`标签，标签内包含内联的脚本代码。

```
<!-- 不好的写法 -->
<script>
    doSomething();
</script>
```

最好将所有的 JavaScript 代码都放入外置文件中，以确保在 HTML 代码中不会有内联的 JavaScript 代码。这样做的原因是出于紧急调试的考虑。当 JavaScript 报错，你的下意识的行为应当是去 JavaScript 文件中查找原因。如果在 HTML 中包含 JavaScript 代码，则会阻断你的工作流（workflow interruption）。你必须要首先确定 JavaScript 代码是在文件里（通常如此）还是在 HTML 中。之后你才会开始调试。

这一点看起来无关紧要，尤其是在有很多优秀且强大的调试工具的今天，但它实际上 是可维护性难题的一个重要方面。“可预见性”（Predictability）会带来更快的调试和开发，并确信（而非猜测）从何入手调试 bug，这会让问题解决得更快、代码总体质量更高。

5.5 将 HTML 从 JavaScript 中抽离

正如我们需要将 JavaScript 从 HTML 中抽离一样，最好也将 HTML 从 JavaScript 中抽离。就像上文提到的，当需要调试一个文本或结构性的问题时，你更希望从 HTML 开始调试。在我的职业生涯中，当我在 HTML 中查找问题原因时，最终却发现真正的问题被深埋在 JavaScript 代码之中，这种情形发生过太多太多次了。

在 JavaScript 中使用 HTML 的情形往往是给`innerHTML`属性赋值时，比如：

```
// 不好的写法
```

```
var div = document.getElementById("my-div");
div.innerHTML = "<h3>Error</h3><p>Invalid e-mail address.</p>";
```

将 HTML 嵌入在 JavaScript 代码中是非常不好的实践。原因有几个。第一，正如上文提到的，它增加了跟踪文本和结构性问题的复杂度。调试上面这段标签的典型方法是首先去浏览器调试工具中的 DOM 树里查找，然后打开页面的 HTML 源码对比其不同。一旦 JavaScript 做了除简单 DOM 操作之外的事情，追踪 bug 就变得很成问题了。

用这种方法（将 JavaScript 从 HTML 中抽离）来解决第二个问题则提高了代码的可维护性。如果你希望修改文本或标签，你只希望去一个地方：可以控制你 HTML 代码的地方。这可能在 PHP 代码中、JSP 文件中或者甚至 Mustache 或 Handlebars 模板中去修改。不管你用了哪种方法，你总是希望你的标签出现在一处，以便很方便地更新它们。嵌入在 JavaScript 代码中的 HTML 标签则不便被修改，因为（通常）你无法（下意识地）想到：如果大部分标签都放置于一个目录下的模板文件里，你何以想到会去 JavaScript 中去修改 HTML 标签呢？

相比于修改 JavaScript 代码，修改标签通常不会引发太多错误。当 HTML 和 JavaScript 混淆在一起时，则将这个问题变得复杂化了。JavaScript 字符串需要对引号做适当的转义，这样做则会导致它和模板语言的原生语法略有差异。

因为多数 Web 应用本质上都是动态的，而在页面的生命周期内，JavaScript 通常用来修改 UI，必然需要通过 JavaScript 向页面插入或修改标签。有多种方法可以以低耦合的方式完成这项工作。

5.5.1 方法 1：从服务器加载

第一种方法是将模板放置于远程服务器^①，使用 XMLHttpRequest 对象来获取外部标签。相比于多页应用（multiple-page applications），这种方法对单页应用（single-page applications）带来更多的便捷。例如，点击一个链接，希望弹出一个新对话框，代码可能如下。

```
function loadDialog(name, oncomplete) {
```

^① 译注：这种方法（从服务器获取模板）很容易造成 XSS 漏洞，需要服务器对模板文件做适当转义处理，比如<和>以及双引号等，当然前端也应当给出与之匹配的渲染规则，总之这种方法需要一揽子前后端的转码和解码策略来尽可能地封堵 XSS 漏洞。

```

var xhr = new XMLHttpRequest();
xhr.open("get", "/js/dialog/" + name, true);

xhr.onreadystatechange = function() {

    if (xhr.readyState == 4 && xhr.status == 200) {

        var div = document.getElementById("dlg-holder");
        div.innerHTML = xhr.responseText;
        oncomplete();

    } else {
        // 处理错误
    }
};

xhr.send(null);
}

```

这里没有将 HTML 字符串嵌入在 JavaScript 里，而是向服务器发起请求获取字符串，这样可以让 HTML 代码以最合适的方式注入到页面中。JavaScript 类库将这个操作做了封装，使得直接给 DOM 元素挂载内容变得非常方便。对此 YUI 和 jQuery 都提供了简单的 API。

```

// YUI
function loadDialog(name, oncomplete) {
    Y.one("#dlg-holder").load("/js/dialog/" + name, oncomplete);
}

// jQuery
function loadDialog(name, oncomplete) {
    $("#dlg-holder").load("/js/dialog/" + name, oncomplete);
}

```

当你需要注入大段 HTML 标签到页面中时，使用远程调用的方式来加载标签是非常有帮助的。出于性能的原因，将大量没用的标签存放于内存或 DOM 中是很糟糕的做法。对于少量的标签段，你可以考虑采用客户端模板。

5.5.2 方法 2：简单客户端模板

客户端模板是一些带“插槽”的标签片段，这些“插槽”会被 JavaScript 程序替换为数据以保证模板的完整可用。比如，一段用来添加数据项的模板看起来就像下面这样。

```
<li><a href="%s">%s</a></li>
```

这段模板中包含%*s* 占位符，这个位置的文本会被程序替换掉（这个格式和 C 语言中的 sprintf()一模一样）。JavaScript 程序会将这些占位符替换为真实数据，然后将结果注入 DOM。下面这段代码给出了这样一个函数及其用法。

```
function sprintf(text) {
    var i=1, args=arguments;
    return text.replace(/%s/g, function() {
        return (i < args.length) ? args[i++] : "";
    });
}

// 用法
var result = sprintf(templateText, "/item/4", "Fourth item");
```

将模板文本传入 JavaScript 是这个过程的重要一环。本质上讲，你一点也不希望在 JavaScript 中嵌入模板文本，而是将模板放置于他处。通常我们将模板定义在其他标签之间，直接存放于 HTML 页面里，这样可以被 JavaScript 读取，用两种方法之一即可做到。第一种方法是在 HTML 注释中包含模板文本。注释是和元素及文本一样的 DOM 节点，因此可以通过 JavaScript 将其提取出来，比如：

```
<ul id="mylist"><!--<li id="item%s"><a href="%s">%s</a></li>-->
<li><a href="/item/1">First item</a></li>
<li><a href="/item/2">Second item</a></li>
<li><a href="/item/3">Third item</a></li>
</ul>
```

在这个用法里，这段注释作为列表的第一个子节点，被恰当地放置于上下文中。下面这段 JavaScript 代码则将模板文本从注释中提取出来。

```
var mylist = document.getElementById("mylist"),
    templateText = mylist.firstChild.nodeValue;
```

一旦提取出了模板文本，紧接着需要将它格式化并插入 DOM。通过下面这个函数可以完成这些操作。

```
function addItem(url, text) {
    var mylist = document.getElementById("mylist"),
        templateText = mylist.firstChild.nodeValue,
        result = sprintf(templateText, url, text);
```

```
        div.innerHTML = result;
        mylist.insertAdjacentHTML("beforeend", result);
    }

// 用法
addItem("/item/4", "Fourth item");
```

我们给这个方法传入了一些数据信息，用它们来处理模板文本，并用 `insertAdjacentHTML()` 将结果注入 HTML。这一步操作将 HTML 字符串转换为一个 DOM 节点，并将它作为子节点插入到``里。

将模板数据嵌入到 HTML 页面里的第二个方法是使用一个带有自定义 `type` 属性的`<script>`元素。浏览器会默认地将`<script>`元素中的内容识别为 JavaScript 代码，但你可以通过给 `type` 赋值为浏览器不识别的类型，来告诉浏览器这不是一段 JavaScript 脚本，比如：

```
<script type="text/x-my-template" id="list-item">
    <li><a href="%s">%s</a></li>
</script>
```

你可以通过`<script>`标签的 `text` 属性来提取模板文本。

```
var script = document.getElementById("list-item"),
    templateText = script.text;
```

这样 `addItem()` 函数就会变为这样。

```
function addItem(url, text) {
    var mylist = document.getElementById("mylist"),
        script = document.getElementById("list-item"),
        templateText = script.text,
        result = sprintf(template, url, text),
        div = document.createElement("div");

    div.innerHTML = result.replace(/^\s*/ , "");
    list.appendChild(div.firstChild);
}

// 用法
addItem("/item/4", "Fourth item");
```

这个版本的函数有一处修改，即去掉了模板文本中的前导空格。之所以会出现这个

多余的前导空格，是因为模板文本总是在<script>起始标签的下一行。如果将模板文本原样注入，则会在<div>里创建一个文本节点，这个文本节点的内容是一个空格，而这个文本节点最终会代替被添加进列表之中。

5.5.3 方法 3：复杂客户端模板

上几节中介绍的模板格式都非常简单，并无太多转义。如果你想用一些更健壮的模板，则可以考虑诸如 Handlebars (<http://handlebarsjs.com/>) 所提供的解决方案。Handlebars 是专为浏览器端 JavaScript 设计的完整的客户端模板系统。

在 Handlebar 的模板中，占位符使用双花括号来表示。我们将上一节中的模板表示为 Handlebars 版本如下。

```
<li><a href="{{url}}">{{text}}</a></li>
```

在 Handlebars 模板中，占位符都标记为一个名称，以便可以在 JavaScript 中设置其映射。Handlebars 建议将模板嵌入 HTML 页面中，并使用 type 属性为 "text/x-handlebars-template" 的<script>标签来表示。

```
<script type="text/x-handlebars-template" id="list-item">
  <li><a href="{{url}}">{{text}}</a></li>
</script>
```

要想使用这个模板，你首先必须将 Handlebars 类库引入页面，这个类库会创建一个名为 Handlebars 的全局变量，用来将模板文本编译为一个函数。

```
var script = document.getElementById("list-item"),
    templateText = script.text,
    template = Handlebars.compile(script.text);
```

这时，变量 template 包含了一个函数，当执行这个函数时则返回一个格式化好的字符串。你需要做的仅仅是传入一个包含 name 和 url 属性的对象。

```
var result = template({
  text: "Fourth item",
  url: "/item/4"
});
```

参数会自动做 HTML 转义，转义操作也是格式化的一部分。转义是为了增强模板

的安全性，并确保简单的文本值不会破坏你的标签结构。比如，字符&会自动转义为&。

现在将它们合并入一个单独的函数中。

```
function addItem(url, text) {
    var mylist = document.getElementById("mylist"),
        script = document.getElementById("list-item"),
        templateText = script.text,
        template = Handlebars.compile(script.text),
        div = document.createElement("div"),
        result;

    result = template({
        text: text,
        url: url
    });

    div.innerHTML = result;
    list.appendChild(div.firstChild);
}

// 用法
addItem("/item/4", "Fourth item");
```

这个简单的例子并未真正体现 Handlebar 的灵活性。除了简单的占位符替换之外，Handlebars 模板同样支持一些简单的逻辑和循环。

假设你想渲染整个列表而非一条记录，但你又想仅在实际存在可渲染的记录时才生成一条记录。你可以像下面这样创建一条 Handlebars 模板。

```
{{#if items}}
<ul>
{{#each items}}
<li><a href="{{url}}">{{text}}</a></li>
{{/each}}
</ul>
{{/if}}
```

在这段模板中，{{#if}}代码段确保了只有 items 数组中至少存在一条记录时才会真正渲染完整的标签。{{#each}}代码段紧接着遍历数组中的每条记录，因此，你将模板编译为一个函数，然后给这个函数传入一个包含 items 属性的对象，比如下面这段代码。

```
// 返回一个空字符串
var result = template({
  items: []
});

// 返回包含两个记录的 HTML 列表
var result = template({
  items: [
    {
      text: "First item",
      url: "/item/1"
    },
    {
      text: "Second item",
      url: "/item/2"
    }
  ]
});
```

Handlebar 包含其他很多种逻辑原语，因此它也是一款强大的 JavaScript 模板引擎。

第6章

避免使用全局变量

JavaScript 执行环境在很多方面都有其独特之处。全局变量和函数的使用便是其中之一。事实上，JavaScript 的初始执行环境是由多种多样的全局变量所定义的，这些全局变量在脚本环境创建之初就已经存在了。我们说这些都是挂载在“全局对象”（global object）上的，“全局对象”是一个神秘的对象，它表示了脚本的最外层上下文。

在浏览器中，`window` 对象往往重载并等同于全局对象，因此任何在全局作用域中声明的变量和函数都是 `window` 对象的属性，比如：

```
var color = "red"

function sayColor() {
    alert(color);
}

console.log(window.color);           // "red"
console.log(typeof window.sayColor); // "function"
```

在这段代码中定义了全局变量 `color` 和全局函数 `sayColor()`，两者都是 `window` 对象的属性，尽管我们并没有显式地执行给 `window` 对象挂载属性的操作。

6.1 全局变量带来的问题

一般来讲，创建全局变量被认为是糟糕的实践，尤其是在团队开发的大背景下更是问题多多。随着代码量的增长，全局变量会导致一些非常重要的可维护性难题。全

局变量越多，引入错误的概率将会因此变得越来越高。

6.1.1 命名冲突

当脚本中的全局变量和全局函数越来越多时，发生命名冲突的概率也随之增高，即很可能无意间就使用了一个已经声明了的变量。所有的变量都被定义为局部变量，这样的代码才是最容易维护的。

比如，回头看一下上个例子中的 `sayColor()` 函数。这个函数直接依赖了全局变量 `color`。如果 `sayColor()` 被定义在外部文件中和 `color` 分离开来，这种依赖关系就很难追踪到了。

```
function sayColor() {  
    alert(color);           // 不好的做法：color 是哪里来的?  
}
```

让我们更进一步，如果 `color` 在脚本中的定义存在有多处，那么随着 `sayColor()` 函数被包含于代码的位置的不同，其执行结果也不尽相同。

全局环境是用来定义 JavaScript 内置对象的地方，如果你给这个作用域添加了自己的变量，接下来则会面临读取浏览器附带的内置变量的风险。比如，对于名字 `color` 来说，绝对是一个不安全的全局变量名。它只是一个普通名词，并没有任何的限定符，因此和浏览器未来的内置 API 或其他开发者的代码产生冲突的概率极高。

6.1.2 代码的脆弱性

一个依赖于全局变量的函数即是深耦合于上下文环境之中。如果环境发生改变，函数很可能就失效了。在上一个例子中，如果全局变量 `color` 不再存在，`sayColor()` 方法将会报错。这意味着任何对全局环境的修改都可能造成某处代码出错。同样，任何函数也会不经意间修改全局变量，导致对全局变量值的依赖变得不稳定。在上个例子中，如果 `color` 被当作参数传入，代码可维护性会变得更佳。

```
function sayColor(color) {  
    alert(color);  
}
```

修改后的这个函数不再依赖于全局变量，因此任何对全局环境的修改都不会影响到

它。由于 `color` 是一个参数，唯一值得注意的是传入函数的值的合法性。其他修改都不会对这个函数完成它本身的任务有任何影响。

当定义函数的时候，最好尽可能多地将数据置于局部作用域内。在函数内定义的任何“东西”都应当采用这种写法。任何来自函数外部的数据都应当以参数形式传进来。这样做可以将函数和其外部环境隔离开来，并且你的修改不会对程序其他部分造成影响。

6.1.3 难以测试

我尝试着在我的第一个大型的 Web 应用中实施一些单元测试。在我即将完成核心框架的搭建时，我加入了一个团队，此后我努力让我的代码变得易于理解以便后续为其执行测试。我非常吃惊地发现，执行测试变成一项极其困难的工序，因为整个框架要依赖于一些全局变量才会正常工作。

任何依赖全局变量才能正常工作的函数，只有为其重新创建完整的全局环境才能正确地测试它。事实上，这意味着你除了要管理全局环境的修改，你还要在两个全局环境中管理它们：生产环境和测试环境。保持两者的同步是很消耗成本的，很快你会发现（代码）可维护性的噩梦才刚刚开始，越到后来越难于理清头绪。

确保你的函数不会对全局变量有依赖，这将增强你的代码的可测试性（testability）。当然，你的函数可能会依赖原生的（native）JavaScript 全局对象，比如 `Date`、`Array` 等。它们是全局环境的一部分，是和 JavaScript 引擎相关的，你的函数总是会用到这些全局对象。总之，为了保证你的代码具有最佳的可测试性，不要让函数对全局变量有依赖^①。

6.2 意外的全局变量

JavaScript 中有不少陷阱，其中一个就是不小心创建全局变量。当你给一个未被 `var` 语句声明过的变量赋值时，JavaScript 就会自动创建一个全局变量。比如：

^① 译注：请注意作者在这里的措辞，表达中分别使用了“全局对象”和“全局变量”，类似 `Date`、`Array` 的全局对象若有必要时是可以直接依赖的，而“全局变量”更多来自程序开发者，而非“原生的”。这里作者要表达的是要避免对“全局变量”的依赖。

```
function doSomething() {  
    var count = 10;  
    title = "Maintainable JavaScript"; // 不好的写法：创建了全局变量  
}
```

这里的代码展示了一个常见的编程错误，即不小心引入了一个全局变量。程序作者的意图是用一个单 `var` 语句声明两个变量，但在第一个变量后不小心敲入了分号而不是逗号，结果就将 `title` 创建成了全局变量。

当你希望创建一个和某个全局变量同名的局部变量时问题稍有复杂。比如，在上一个例子中，如果存在一个名叫 `count` 的全局变量，那么它在函数中将会被局部变量 `count` “遮盖” (shadowed)。这时函数内对 `count` 的操作都是基于局部变量的 `count`，除非你显式地使用 `window.count` 来访问全局变量。这样的程序是 OK 的。

不小心省略 `var` 语句可能意味着你在不知情的情况下修改某个已存在的全局变量。看一下这段代码。

```
function doSomething() {  
    var count = 10;  
    name = "Nicholas"; // 不好的写法：创建了全局变量  
}
```

这个例子中的错误更加过分，因为 `name` 实际上是 `window` 的一个默认属性。`window.name` 属性经常用于框架 (frame) 和 iframe 的场景中，当点击链接时，可以通过指定打开链接的目标容器来控制其在特定的框架或选项卡^① 中显示，不小心修改 `name` 会影响到站点的链接导航^②。

最好的规则就是总是使用 `var` 来定义变量，哪怕是定义全局变量。这样做能大大降低某些场景里省略 `var` 所导致错误的可能性。

避免意外的全局变量

当你不小心创建了全局变量时 JavaScript 并不会报任何警告。这时就需要一些诸如 JSLint 和 JSHint 的工具来发挥作用了。如果你给一个未声明的变量赋值，这两个工具都会报警告。比如：

```
foo = 10;
```

① 译注：“选项卡”指浏览器的标签页。

② 译注：指链接打开方式。

JSLint 和 JSHint 扫描到这行代码时都会报“‘foo’未定义”，提醒你变量 foo 并未通过 var 语句来声明。

当你不小心修改了全局变量的值，JSLint 和 JSHint 也都会非常聪明地给出提示。在那个重写 name 值的例子中，JSLint 和 JSHint 都会报警告：这个变量是只读的。实际上，这个变量并不是只读的，但我们应当将它当只读变量来对待，因为修改内置全局变量往往会导致错误。如果你试图给其他全局对象（比如 window 和 document）赋值，同样也会报警告。

严格模式下，解析和执行 JavaScript 的规则发生了变化，也为这个问题提供了解决方案。只要在函数顶部加入了“`user strict`”，则通知 JavaScript 引擎在运行代码前执行更严格的错误处理和语法检查。其中一个规则的变化就是（严格模式）可以探测未声明变量的赋值操作。一旦发生这种情形，JavaScript 引擎会报错。比如：

```
"use strict";
foo = 10; // 引用错误: foo 未被定义
```

如果你在支持严格模式的环境（IE 10+、FireFox 4+、Safari 5.1+、Opera 12+、或 Chrome）中运行这段代码，第 2 行将会抛出一个 ReferenceError，报错信息是“foo 未被定义”。

将代码置于严格模式将会改变很多 JavaScript 的行为，如果你在处理老的代码时则要非常小心地使用严格模式。对于新的代码，最好总是使用严格模式来避免意外的全局变量，同时其他一些常见的编程错误也能在严格模式下被捕捉到。

6.3 单全局变量方式

你看到这个标题可能会问：“写 JavaScript 程序怎么可能不需要引入全局变量？”尽管使用某些小巧的编码模式可以在技术上实现，但长期看来这种实现方式的可行性和可维护性都欠佳。当涉及到团队开发时，这往往意味着要在不同的场景下加载不同的文件，而这些被分割的代码之间能够相互通信的唯一方式就是所有代码都（对某个全局变量）有相同的依赖。最佳方法是依赖尽可能少的全局变量^①，即只创建一个全局变量。

单全局变量模式已经在各种流行的 JavaScript 类库中广泛使用了。

^① 译注：这个全局变量必须是有其特征，而不会与其他任何全局变量产生冲突的。

- YUI 定义了唯一一个 YUI 全局对象。
- jQuery 定义了两个全局对象，\$和jQuery。只有在\$被其他的类库使用了的情况下，为了避免冲突，应当使用jQuery。
- Dojo 定义了一个 dojo 全局对象。
- Closure 类库定义了一个 goog 全局对象。

“单全局变量”的意思是所创建的这个唯一全局对象名是独一无二的（不会和内置 API 产生冲突），并将你所有的功能代码都挂载到这个全局对象上。因此每个可能的全局变量都成为你唯一全局对象的属性，从而不会创建多个全局变量。比如，假设我想让一个对象表示本书的一章，代码看起来会是像下面这样。

```
function Book(title) {  
    this.title = title;  
    this.page = 1;  
}  
  
Book.prototype.turnPage = function(direction) {  
    this.page += direction;  
};  
  
var Chapter1 = new Book("Introduction to Style Guidelines");  
var Chapter2 = new Book("Basic Formatting");  
var Chapter3 = new Book("Comments");
```

这段代码创建了 4 个全局对象：Book、Chapter1、Chapter2 和 Chapter3。单全局变量模式则只会创建一个全局对象并将这些对象都赋值为它的属性。

```
var MaintainableJS = {};  
  
MaintainableJS.Book = function(title) {  
    this.title = title;  
    this.page = 1;  
};  
  
MaintainableJS.Book.prototype.turnPage = function(direction) {  
    this.page += direction;  
};  
  
MaintainableJS.Chapter1 = new MaintainableJS.Book("Introduction to Style  
Guidelines");
```

```
MaintainableJS.Chapter2 = new MaintainableJS.Book("Basic Formatting");
MaintainableJS.Chapter3 = new MaintainableJS.Book("Comments");
```

这段代码只有一个全局对象，即 `MaintainableJS`，其他任何信息都挂载到这个对象上。因为团队中的每个人都知道这个全局对象，因此很容易做到继续为它添加属性以避免全局污染。

6.3.1 命名空间

即使你的代码只有一个全局对象，也存在着全局污染的可能性。大多数使用单全局变量模式的项目同样包含“命名空间”的概念。命名空间是简单的通过全局对象的单一属性表示的功能性分组。比如，`YUI` 就是依照命名空间的思路来管理其代码的。`Y.DOM` 下的所有方法都是和 DOM 操作相关的，`Y.Event` 下的所有方法都是和事件相关的，以此类推。

将功能按照命名空间进行分组，可以让你的单全局对象变得井然有序，同时可以让团队成员能够知晓新功能应该属于哪个部分，或者知道去哪里查找已有的功能。当我在 `Yahoo!` 工作的时候，就有一个不成文的约定，即每个站点都将自己的命名空间都挂载至 `Y` 对象上，因此 “`My Yahoo!`” 使用 `Y.My`，邮箱使用 `Y.Mail`，等等。这样团队成员则可以放心大胆地使用其他人的代码，而不必担心产生冲突。

在 JavaScript 中你可以使用对象来轻而易举地创建你自己的命名空间，比如：

```
var ZakasBooks = {};
// 表示这本书的命名空间
ZakasBooks.MaintainableJavaScript = {};
// 表示另外一本书的命名空间
ZakasBooks.HighPerformanceJavaScript = {}
```

一个常见的约定是每个文件中都通过创建新的全局对象来声明自己的命名空间。在这种情况下，上面这个例子给出的方法是够用的。

同样有另外一些场景，每个文件都需要给一个命名空间挂载东西。在这种情况下，你需要首先保证这个命名空间是已经存在的。这时全局对象非破坏性的处理命名空间的方式则变得非常有用。完成这项操作的基本模式是像下面这样的。

```

var YourGlobal = {
    namespace: function(ns) {
        var parts = ns.split("."),
            object = this,
            i, len;

        for (i=0, len=parts.length; i < len; i++) {
            if (!object[parts[i]]) {
                object[parts[i]] = {};
            }
            object = object[parts[i]];
        }

        return object;
    }
};

```

变量 `YourGlobal` 实际上可以表示任意名字。最重要的部分在于 `namespace()` 方法，我们给这个方法传入一个表示命名空间对象的字符串，它会非破坏性地（`nondestructively`）创建这个命名空间。基本的用法如下。

```

/*
 * 同时创建 YourGlobal.Books 和 YourGlobal.Books.MaintainableJavaScript
 * 因为之前没有创建过它们，因此每个都是全新创建的
 */
YourGlobal.namespace("Books.MaintainableJavaScript");

// 现在你可以使用这个命名空间
YourGlobal.Books.MaintainableJavaScript.author = "Nicholas C. Zakas";

/*
 * 不会操作 YourGlobal.Books 本身，同时会给它添加 HighPerformanceJavaScript
 * 它会保持 YourGlobal.Books.MaintainableJavaScript 原封不动
 */
YourGlobal.namespace("Books.HighPerformanceJavaScript");

// 仍然是合法的引用
console.log(YourGlobal.Books.MaintainableJavaScript.author);

// 你同样可以在方法调用之后立即给它添加新属性
YourGlobal.namespace("Books").ANewBook = {};

```

基于你的单全局对象使用 `namespace()` 方法可以让开发者放心地认为命名空间总是存在的。这样，每个文件都可以首先调用 `namespace()` 来声明开发者将要使用的命名空间，这样做不会对已有的命名空间造成任何破坏。这个方法可以让开发者解放

出来，在使用命名空间之前不必再去判断它是否存在。

由于你的代码不是独立存在的，因此要围绕命名空间定义一些约定。是否应该以首字母大写的形式来定义命名空间，就像 YUI？还是都用小写字母形式来定义命名空间，就像 Dojo？这个是个人喜好问题，但是首先定义这些约定可以让后续的团队成员在使用单全局变量时更加高效。

6.3.2 模块

另外一种基于单全局变量的扩充方法是使用模块（modules）。模块是一种通用的功能片段，它并没有创建新的全局变量或命名空间。相反，所有的这些代码都存放于一个表示执行一个任务或发布一个接口的单函数中。可以用一个名称来表示这个模块，同样这个模块可以依赖其他模块。

JavaScript 本身不包含正式的模块概念，自然也没有模块语法（至少到 ECMAScript6 中都看不到这个迹象），但的确有一些通用的模式用来创建模块。两种最流行的是“YUI 模块”模式和“异步模块定义”（Asynchronous Module Definition，简称 AMD）模式。

YUI 模块

从字面含义理解，YUI 模块就是使用 YUI JavaScript 类库来创建新模块的一种模式。YUI3 中包含了模块的概念，写法如下。

```
YUI.add("module-name", function(Y) {  
    // 模块正文  
}, "version", { requires: [ "dependency1", "dependency2" ] });
```

我们通过调用 YUI.add() 并给它传入模块名字、待执行的函数（被称作工厂方法）和可选的依赖列表来添加 YUI 模块。“模块正文”处则是你写所有的模块代码的地方。参数 Y 是 YUI 的一个实例，这个实例包含所有依赖的模块提供的内容。YUI 中约定在每个模块内使用命名空间的方式来管理模块代码，比如：

```
YUI.add("my-books", function(Y) {  
    // 添加一个命名空间
```

```
Y.namespace("Books.MaintainableJavaScript");

Y.Books.MaintainableJavaScript.author = "Nicholas C. Zakas";
}, "1.0.0", { requires: [ "dependency1", "dependency2" ] });


```

同样，依赖也是以 Y 对象命名空间的形式传入进来。因此 YUI 实际上是将命名空间和模块的概念合并在了一起，总体上提供一种灵活的解决方案^①。

通过调用 YUI().use()函数并传入想加载的模块名称来使用你的模块。

```
YUI().use("my-books", "another-module", function(Y) {

    console.log(Y.Books.MaintainableJavaScript.author);

});
```

这段代码以加载名叫“my-books”和“another-module”的两个模块开始，YUI 会确保这些模块的依赖都会完全加载完成，然后执行模块的正文代码，最后才会执行传入 YUI().use()的回调函数。回调函数会带回 Y 对象，Y 对象里包含了加载的模块对它做的修改，这时你的应用代码就可以放心地执行了。

想要了解更多关于 YUI 模块的代码，可以参照 YUI 的文档 <http://yuilibrary.com/yui/docs/yui> “异步模块定义”（AMD）

AMD 模块和 YUI 模块有诸多相似之处。你指定模块名称、依赖和一个工厂方法，依赖加载完成后执行这个工厂方法。这些内容全部作为参数传入一个全局函数 define()中，其中第一个参数是模块名称，然后是依赖列表，最后是工厂方法。AMD 模块和 YUI 模块最大的不同在于，(AMD 中)每一个依赖都会对应到独立的参数传入工厂方法里，比如：

```
define("module-name", [ "dependency1", "dependency2" ],
function(dependency1, dependency2) {

    // 模块正文

});
```

^① 译注：YUI 所提供的这种解决方案实际上是综合了很多因素的结果，命名空间和模块语法只是其中的一部分，其他还包括 YUI 的模块加载器、多文件合并策略和多版本冲突等诸多方面。

因此，每个被命名的依赖最后都会创建一个对象，这个对象会被带入工厂方法中。AMD 以这种方式来尝试避免命名冲突，因为直接在模块中使用命名空间有可能发生命名冲突。和 YUI 模块中创建新的命名空间的方法不同，AMD 模块则期望从工厂方法中返回它们的公有接口，比如：

```
define("my-books", [ "dependency1", "dependency2" ],
       function(dependency1, dependency2) {
         var Books = {};
         Books.MaintainableJavaScript = {
           author: "Nicholas C. Zakas"
         };
         return Books;
     });

```

AMD 模块同样可以是匿名的，完全省略模块名称。因为模块加载器可以将 JavaScript 文件名当作模块名称。所以如果你有一个名叫 my-books.js 的文件，你的模块可以只通过模块加载器来加载，你可以像这样定义你的模块。

```
define([ "dependency1", "dependency2" ],
       function(dependency1, dependency2) {
         var Books = {};
         Books.MaintainableJavaScript = {
           author: "Nicholas C. Zakas"
         };
         return Books;
     });

```

AMD 模块规范给出的定义模块的方式只有很少的几个选项。想要了解更多内容，请参照 AMD 规范：<https://github.com/amdjs/amdjs-api/wiki/AMD>。

要想使用 AMD 模块，你需要使用一个与之兼容的模块加载器。Dojo 的标准模块加载器支持 AMD 模块的加载，因此你可以像下面这样来加载“my-books”模块。

```
// 使用 Dojo 加载 AMD 模块
var books = dojo.require("my-books");
console.log(books.MaintainableJavaScript.author);
```

Dojo 同样将自己也封装为 AMD 模块，叫做“dojo”，因此它也可以被其他 AMD 模块加载。

另一个模块加载器是 RequireJS (<http://www.requirejs.org/>)。RequireJS 添加了另一个全局函数 require()，专门用来加载指定的依赖和执行回调函数，比如：

```
// 使用 RequireJS 加载 AMD 模块
require([ "my-book" ], function(books) {
    console.log(books.MaintainableJavaScript.author);
});
```

调用 require() 时会首先立即加载依赖，这些依赖都加载完成后会即刻执行回调函数（和 YUI().use() 类似）。

RequireJS 模块加载器包含很多内置逻辑来让模块的加载更加方便，包括名字到目录的对应表以及多语种选项（internationalization options.）。jQuery 和 Dojo 都可以使用 RequireJS 来加载 AMD 模块^①。

6.4 零全局变量

你的 JavaScript 代码注入到页面时是可以做到不用创建全局变量的。这种方法应用场景不多，因此只有在某些特殊场景下才会有用。最常见的情形就是一段不会被其他脚本访问到的完全独立的脚本。之所以存在这种情形，是因为所有所需的脚本都会合并到一个文件，或者因为这段非常短小且不提供任何接口的代码会被插入至一个页面中。最常见的用法是创建一个书签。

书签是独立的，它们并不知晓页面中包含什么且不需要页面知道它的存在。最终我们需要一段“零全局变量”的脚本嵌入到页面中，实现方法就是使用一个立即执行的函数调用并将所有脚本放置其中，比如：

```
(function(win) {
    var doc = win.document;
```

① 译注：和 AMD 规范类似的还有 CommonJS 规范，CommonJS 规范更适用于服务器端或纯脚本编程，而在浏览器端则不大适用。

```
// 在这里定义其他的变量  
// 其他相关代码  
} (window));
```

这段立即执行的代码传入了 window 对象，因此这段代码不需要直接引用任何全局变量。在这个函数内部，变量 doc 是指向 document 对象的引用，只要是函数代码中没有直接修改 window 或 doc 且所有变量都是用 var 关键字来定义，这段脚本则可以注入到页面中而不会产生任何全局变量。之后你可以通过将函数设置为严格模式（strict mode）来避免创建全局变量（当然，首先要浏览器支持严格模式）。

```
(function (win) {  
    "use strict";  
  
    var doc = win.document;  
  
    // 在这里定义其他的变量  
    // 其他相关代码  
} (window));
```

这个函数包装器（function wrapper）可以用于任何不需要创建全局对象的场景。正如上文提到的，这种模式的使用场景有限。只要你的代码需要被其他的代码所依赖，就不能使用这种零全局变量的方式。如果你的代码需要在运行时被不断扩展或修改也不能使用零全局变量的方式。但是，如果你的脚本非常短，且不需要和其他代码产生交互，可以考虑使用零全局变量的方式来实现代码。

事件处理

在所有 JavaScript 应用中事件处理都是非常重要的。所有的 JavaScript 均通过事件绑定到 UI 上，所以大多数前端工程师需要花费很多时间来编写和修改事件处理程序。遗憾的是，在 JavaScript 诞生之初，这部分内容并未受太多重视。甚至当开发者们开始热衷于将传统的软件架构概念融入到 JavaScript 里时，事件绑定仍然没有受到多大重视。大多数事件处理相关的代码和事件环境（对于开发者来说，每次事件触发时才会可用）紧紧耦合在一起，导致可维护性很糟糕。

7.1 典型用法

多数开发者都很了解，当事件触发时，事件对象（`event` 对象）会作为回调参数传入事件处理程序中。`event` 对象包含所有和事件相关的信息，包括事件的宿主（`target`）以及其他和事件类型相关的数据。鼠标事件会将其位置信息暴露在 `event` 对象上，键盘事件会将按键的信息暴露在 `event` 对象上，触屏事件会将触摸位置和持续时间（`duration`）暴露在 `event` 对象上。只有提供了所有这些信息，UI 才会正确地执行交互。

在很多场景中，你只是用到了 `event` 所提供信息的一小部分，看下面这段代码。

```
// 不好的写法
function handleClick(event) {
    var popup = document.getElementById("popup");
    popup.style.left = event.clientX + "px";
    popup.style.top = event.clientY + "px";
```

```
popup.className = "reveal";
}

// 第 7 章中的 addListener()
addListener(element, "click", handleClick);
```

这段代码只用到了 `event` 对象的两个属性：`clientX` 和 `clientY`。在将元素显示在页面里之前先用这两个属性给它作定位。尽管这段代码看起来非常简单且没有什么问题，但实际上这是不好的写法，因为这种做法有其局限性。

7.2 规则 1：隔离应用逻辑

上段实例代码的第一个问题是事件处理程序包含了应用逻辑（application logic）。应用逻辑是和应用相关的功能性代码，而不是和用户行为相关的。上段实例代码中，应用逻辑是在特定位置显示一个弹出框。尽管这个交互应当是在用户点击某个特定元素时发生，但情况并不总是如此。

将应用逻辑从所有事件处理程序中抽离出来的做法是一种最佳实践，因为说不定什么时候其他地方就会触发同一段逻辑。比如，有时你需要在用户将鼠标移到某个元素上时判断是否显示弹出框，或者当按下键盘上的某个键时也作同样的逻辑判断。这样多个事件的处理程序执行了同样的逻辑，而你的代码却被不小心复制了多份。

将应用逻辑放置于事件处理程序中的另一个缺点是和测试有关的。测试时需要直接触发功能代码，而不必通过模拟对元素的点击来触发。如果将应用逻辑放置于事件处理程序中，唯一的测试方法是制造事件的触发。尽管某些测试框架可以模拟触发事件，但实际上这不是测试的最佳方法。调用功能性代码最好的做法就是单个的函数调用。

你总是需要将应用逻辑和事件处理的代码拆分开来。如果要对上一段实例代码进行重构，第一步是将处理弹出框逻辑的代码放入一个单独的函数中，这个函数很可能挂载于为该应用定义的一个全局对象上。事件处理程序应当总是在一个相同的全局对象中，因此就有了以下两个方法。

```
// 好的写法 - 拆分应用逻辑
var MyApplication = {
```

```

        handleClick: function(event) {
            this.showPopup(event);
        },

        showPopup: function(event) {
            var popup = document.getElementById("popup");
            popup.style.left = event.clientX + "px";
            popup.style.top = event.clientY + "px";
            popup.className = "reveal";
        }
    };

    addListener(element, "click", function(event) {
        MyApplication.handleClick(event);
   });

```

之前在事件处理程序中包含的所有应用逻辑现在转移到了 `MyApplication.showPopup()` 方法中。现在 `MyApplication.handleClick()` 方法只做一件事情，即调用 `MyApplication.showPopup()`。若应用逻辑被剥离出去，对同一段功能代码的调用可以在多点发生，则不需要一定依赖于某个特定事件的触发，这显然更加方便。但这只是拆解事件处理程序代码的第一步。

7.3 规则 2：不要分发事件对象

在剥离开应用逻辑之后，上段实例代码还存在一个问题，即 `event` 对象被无节制地分发。它从匿名的事件处理函数传入了 `MyApplication.handleClick()`，然后又传入了 `MyApplication.showPopup()`。正如上文提到的，`event` 对象上包含很多和事件相关的额外信息，而这段代码只用到了其中的两个而已。

应用逻辑不应当依赖于 `event` 对象来正确完成功能，原因如下。

- 方法接口并没有表明哪些数据是必要的。好的 API 一定是对于期望和依赖都是透明的^①。将 `event` 对象作为参数并不能告诉你 `event` 的哪些属性是有用的，用来干什么？

^① 译注：作者的意思是说好的 API 要更明确清楚，但这个观点我们需要辩证地来对待，如果明确知道回调传值的用处以及需要传哪些值，当然更好，但更多的时候我们并不知道应用逻辑做何种事情，因此需要为应用逻辑提供尽可能多的信息，如何利用这些信息，效率如何统统交由应用逻辑负责，以达到某种层次的解偶。译者认为这个原因是次要的，主要原因是作者提到的第二点。

- 因此，如果你想测试这个方法，你必须重新创建一个 `event` 对象并将它作为参数传入。所以，你需要确切地知道这个方法使用了哪些信息，这样才能正确地写出测试代码。

这些问题^①在大型 Web 应用中都是不可取的。代码不够明晰就会导致 bug。

最佳的办法是让事件处理程序使用 `event` 对象来处理事件，然后拿到所有需要的数据传给应用逻辑。例如，`MyApplicaiton.showPopup()`方法只需要两个数据，`x` 坐标和`y`坐标。这样我们将方法重写一下，让它来接收这两个参数。

```
// 好的写法
var MyApplication = {

    handleClick: function(event) {
        this.showPopup(event.clientX, event.clientY);
    },

    showPopup: function(x, y) {
        var popup = document.getElementById("popup");
        popup.style.left = x + "px";
        popup.style.top = y + "px";
        popup.className = "reveal";
    }

};

addListener(element, "click", function(event) {
    MyApplication.handleClick(event); // 可以这样用
});
```

在这段新重写的代码中，`MyApplication.handleClick()`将 `x` 坐标和 `y` 坐标传入了`MyApplication.showPopup()`，代替了之前传入的事件对象。可以很清晰地看到`MyApplication.showPopup()`所期望传入的参数，并且在测试或代码的任意位置都可以很轻易地直接调用这段逻辑，比如：

```
// 这样调用非常棒
MyApplication.showPopup(10, 10);
```

当处理事件时，最好让事件处理程序成为接触到 `event` 对象的唯一的函数。事件处理程序应当在进入应用逻辑之前针对 `event` 对象执行任何必要的操作，包括阻止默

① 译注：指接口格式不清晰和自行构造 `event` 对象来用于测试。

认事件或阻止事件冒泡，都应当直接包含在事件处理程序中。比如：

```
// 好的做法
var MyApplication = {

    handleClick: function(event) {

        // 假设事件支持 DOM Level2
        event.preventDefault();
        event.stopPropagation();

        // 传入应用逻辑
        this.showPopup(event.clientX, event.clientY);
    },

    showPopup: function(x, y) {
        var popup = document.getElementById("popup");
        popup.style.left = x + "px";
        popup.style.top = y + "px";
        popup.className = "reveal";
    }
};

addListener(element, "click", function(event) {
    MyApplication.handleClick(event); // 可以这样做
});
```

在这段代码中，MyAPplication.handleClick()是事件处理程序，因此它在将数据传入应用逻辑之前调用了 event.preventDefault()和 event.stopPropagation()，这清楚地展示了事件处理程序和应用逻辑之间的分工。因为应用逻辑不需要对 event 产生依赖，进而在很多地方都可以轻松地使用相同的业务逻辑，包括写测试代码。

第8章

避免“空比较”

在 JavaScript 中，我们常常会看到这种代码：变量与 null 的比较（这种用法很有问题），用来判断变量是否被赋予了一个合理的值。比如：

```
var Controller = {
    process: function(items) {
        if (items !== null) { // 不好的写法
            items.sort();
            items.forEach(function(item) {
                // 执行一些逻辑
            });
        }
    }
};
```

在这段代码中，process()方法显然希望 items 是一个数组，因为我们看到 items 拥有 sort()和 forEach()。这段代码的意图非常明显：如果参数 items 不是一个数组，则停止接下来的操作。这种写法的问题在于，和 null 的比较并不能真正避免错误的发生。items 的值可以是 1，也可以是字符串，甚至可以是任意对象。这些值都和 null 不相等，进而会导致 process()方法一旦执行到 sort()时就会出错。

仅仅和 null 比较并不能提供足够的信息来判断后续代码的执行是否真的安全。好在 JavaScript 为我们提供了多种方法来检测变量的真实值。

8.1 检测原始值

在 JavaScript 中有 5 种原始类型：字符串、数字、布尔值、null 和 undefined。如果

你希望一个值是字符串、数字、布尔值或 `undefined`，最佳选择是使用 `typeof` 运算符。`typeof` 运算符会返回一个表示值的类型的字符串。

- 对于字符串，`typeof` 返回 “`string`”。
- 对于数字，`typeof` 返回 “`number`”。
- 对于布尔值，`typeof` 返回 “`boolean`”。
- 对于 `undefined`，`typeof` 返回 “`undefined`”。

`typeof` 的基本语法是：

```
typeof variable
```

你也可以这样来用 `typeof`：

```
typeof(variable)
```

尽管这是合法的 JavaScript 语法，这种用法让 `typeof` 看起来像一个函数而非运算符。鉴于此，我们更推荐无括号的写法。

使用 `typeof` 来检测这 4 种原始值类型是非常安全的做法。来看下面一些例子。

```
// 检测字符串
if (typeof name === "string") {
    anotherName = name.substring(3);
}

// 检测数字
if (typeof count === "number") {
    updateCount(count);
}

// 检测布尔值
if (typeof found === "boolean" && found) {
    message("Found!");
}

// 检测 undefined
if (typeof MyApp === "undefined") {
    MyApp = {
```

```
// 其他的代码  
};  
}  
}
```

`typeof` 运算符的独特之处在于，将其用于一个未声明的变量也不会报错。未定义的变量和值为 `undefined` 的变量通过 `typeof` 都将返回 “`undefined`”。

最后一个原始值，`null`，一般不应用于检测语句。正如上文提到的，简单地和 `null` 比较通常不会包含足够的信息以判断值的类型是否合法。但有一个例外，如果所期望的值真的是 `null`，则可以直接和 `null` 进行比较。这时应当使用`==`或者`!=`来和 `null` 进行比较，比如：

```
// 如果你需要检测 null，则使用这种方法  
var element = document.getElementById("my-div");  
if (element != null) {  
    element.className = "found";  
}
```

如果 DOM 元素不存在，则通过 `document.getElementById()` 得到的值为 `null`。这个方法要么返回一个节点，要么返回 `null`。由于这时 `null` 是可预见的一种输出，则可以使用`!=`来检测返回结果。

运行 `typeof null` 则返回 “`object`”，这是一种低效的判断 `null` 的方法。如果你需要检测 `null`，则直接使用恒等运算符 (`==`) 或非恒等运算符 (`!=`)。

8.2 检测引用值

引用值也称作对象 (`object`)。在 JavaScript 中除了原始值之外的值都是引用。有这样几种内置的引用类型：`Object`、`Array`、`Date` 和 `Error`，数量不多。`typeof` 运算符在判断这些引用类型时则显得力不从心，因为所有对象都会返回 “`object`”。

```
console.log(typeof {});           // "object"  
console.log(typeof []);           // "object"  
console.log(typeof new Date());   // "object"  
console.log(typeof new RegExp()); // "object"
```

`typeof` 另外一种不推荐的用法是当检测 `null` 的类型时，`typeof` 运算符用于 `null` 时将会返回 “`object`”。

```
console.log(typeof null); // "object"
```

这看上去很怪异，被认为是标准规范的严重 bug，因此在编程时要杜绝使用 `typeof` 来检测 `null` 的类型。

检测某个引用值的类型的最好方法是使用 `instanceof` 运算符。`instanceof` 的基本语法是：

```
value instanceof constructor
```

这里是一些例子。

```
// 检测日期
if (value instanceof Date) {
    console.log(value.getFullYear());
}

// 检测正则表达式
if (value instanceof RegExp) {
    if (value.test(anotherValue)) {
        console.log("Matches");
    }
}

// 检测 Error
if (value instanceof Error) {
    throw value;
}
```

`instanceof` 的一个有意思的特性是它不仅检测构造这个对象的构造器，还检测原型链。原型链包含了很多信息，包括定义对象所采用的继承模式。比如，默认情况下，每个对象都继承自 `Object`，因此每个对象的 `value instanceof Object` 都会返回 `true`。比如：

```
var now = new Date();

console.log(now instanceof Object); // true
console.log(now instanceof Date); // true
```

因为这个原因，使用 `value instanceof Object` 来判断对象是否属于某个特定类型的做法并非最佳。

`instanceof` 运算符也可以检测自定义的类型，比如：

```
function Person(name) {  
    this.name = name;  
}  
  
var me = new Person("Nicholas");  
  
console.log(me instanceof Object); // true  
console.log(me instanceof Person); // true
```

这段示例代码中创建了 `Person` 类型。变量 `me` 是 `Person` 的实例，因此 `me instanceof Person` 是 `true`。上文也提到，所有的对象都被认为是 `Object` 的实例，因此 `me instanceof Object` 也是 `true`。

在 JavaScript 中检测自定义类型时，最好的做法就是使用 `instanceof` 运算符，这也是唯一的方法。同样对于内置 JavaScript 类型也是如此（使用 `instanceof` 运算符）。但是，有一个严重的限制。

假设一个浏览器帧（frame A）里的一个对象被传入到另一个帧（frame B）中。两个帧里都定义了构造函数 `Person`。如果来自帧 A 的对象是帧 A 的 `Person` 的实例，则如下规则成立。

```
// true  
frameAPersonInstance instanceof frameAPerson  
  
// false  
frameAPersonInstance instanceof frameBPerson
```

因为每个帧（frame）都拥有 `Person` 的一份拷贝，它被认为是该帧（frame）中的 `Person` 的拷贝的实例，尽管两个定义可能是完全一样的。

这个问题不仅出现在自定义类型身上，其他两个非常重要的内置类型也有这个问题：函数和数组。对于这两个类型来说，一般用不着使用 `instanceof`。

8.2.1 检测函数

从技术上讲，JavaScript 中的函数是引用类型，同样存在 `Function` 构造函数，每个函数都是其实例，比如：

```
function myFunc() {}

// 不好的写法
console.log(myFunc instanceof Function); // true
```

然而，这个方法亦不能跨帧（frame）使用，因为每个帧都有各自的 Function 构造函数。好在 typeof 运算符也是可以用于函数的，返回“function”。

```
function myFunc() {}

// 好的写法
console.log(typeof myFunc === "function"); // true
```

检测函数最好的方法是使用 typeof，因为它可以跨帧（frame）使用。

用 typeof 来检测函数有一个限制。在 IE 8 和更早版本的 IE 浏览器中，使用 typeof 来检测 DOM 节点（比如 document.getElementById()）中的函数都返回“object”而不是“function”。比如：

```
// IE 8 及其更早版本的 IE
console.log(typeof document.getElementById); // "object"
console.log(typeof document.createElement); // "object"
console.log(typeof document.getElementsByTagName); // "object"
```

之所以出现这种怪异的现象是因为浏览器对 DOM 的实现有差异。简言之，这些早版本的 IE 并没有将 DOM 实现为内置的 JavaScript 方法，导致内置 typeof 运算符将这些函数识别为对象。因为 DOM 是有明确定义的，了解到对象成员如果存在则意味着它是一个方法，开发者往往通过 in 运算符来检测 DOM 的方法，比如：

```
// 检测 DOM 方法
if ("querySelectorAll" in document) {
    images = document.querySelectorAll("img");
}
```

这段代码检查 querySelectorAll 是否定义在了 document 中，如果是，则使用这个方法。尽管不是最理想的方法，如果想在 IE 8 及更早浏览器中检测 DOM 方法是否存在，这是最安全的做法。在其他所有的情形中，typeof 运算符是检测 JavaScript 函数的最佳选择。

8.2.2 检测数组

JavaScript 中最古老的跨域问题之一就是在帧（frame）之间来回传递数组。开发者很快发现 `instanceof Array` 在此场景中不总是返回正确的结果。正如上文提到的，每个帧（frame）都有各自的 `Array` 构造函数，因此一个帧（frame）中的实例在另外一个帧里不会被识别。Douglas Crockford 首先推荐使用“鸭式辨型”（duck typing）^① 来检测其 `sort()` 方法是否存在。

```
// 采用鸭式辨型的方法检测数组
function isArray(value) {
    return typeof value.sort === "function";
}
```

这种检测方法依赖一个事实，即数组是唯一包含 `sort()` 方法的对象。当然，如果传入 `isArray()` 的参数是一个包含 `sort()` 方法的对象，它也会返回 `true`。

关于如何在 JavaScript 中检测数组类型已经有很多研究了，最终，Juriy Zaytsev（也被称作 Kangax）给出了一种优雅的解决方案。

```
function isArray(value) {
    return Object.prototype.toString.call(value) === "[object Array]";
}
```

Kangax 发现调用某个值的内置 `toString()` 方法在所有浏览器中都会返回标准的字符串结果。对于数组来说，返回的字符串为 “[object Array]”，也不用考虑数组实例是在哪个帧（frame）中被构造出来的。Kangax 给出的解决方案很快流行起来，并被大多数 JavaScript 类库所采纳。

这种方法在识别内置对象时往往十分有用，但对于自定义对象请不要用这种方法。比如，内置 `JSON` 对象使用这种方法将返回 “[object JSON]”。

从那时起，ECMAScript5 将 `Array.isArray()` 正式引入 JavaScript。唯一的目的是准确地检测一个值是否为数组。同 Kangax 的函数一样，`Array.isArray()` 也可以检测跨帧（frame）传递的值，因此很多 JavaScript 类库目前都类似地实现了这个方法。

^① 译注：“鸭式辨型”是由作家 James Whitcomb Riley 首先提出的概念，即“像鸭子一样走路、游泳并且嘎嘎叫的鸟就是鸭子”，本质上是关注“对象能做什么”，而不要关注“对象是什么”，更多内容请参照《JavaScript 权威指南》（第六版）9.5.4 小节。

```
function isArray(value) {
    if (typeof Array.isArray === "function") {
        return Array.isArray(value);
    } else {
        return Object.prototype.toString.call(value) === "[object Array]";
    }
}
```

IE 9+、Firefox 4+、Safari 5+、Opera 10.5+和 Chrome 都实现了 `Array.isArray()` 方法。

8.3 检测属性

另外一种用到 `null`(以及 `undefined`)的场景是当检测一个属性是否在对象中存在时，比如：

```
// 不好的写法：检测假值
if (object[property_name]) {
    // 一些代码
}

// 不好的写法：和 null 相比较
if (object[property_name] != null) {
    // 一些代码
}

// 不好的写法：和 undefined 比较
if (object[property_name] != undefined) {
    // 一些代码
}
```

上面这段代码里的每个判断，实际上是通过给定的名字来检查属性的值，而非判断给定的名字所指的属性是否存在，因为当属性值为假值（*falsy value*）时结果会出错，比如 `0`、`""`（空字符串）、`false`、`null` 和 `undefined`。毕竟，这些都是属性的合法值。比如，如果属性记录了一个数字，则这个值可以是零。这样的话，上段代码中的第一个判断就会导致错误。以此类推，如果属性值为 `null` 或者 `undefined` 时，三个判断都会导致错误。

判断属性是否存在的最好的方法是使用 `in` 运算符。`in` 运算符仅仅会简单地判断属性是否存在，而不会去读属性的值，这样就可以避免出现本小节中前文提到的有歧义的语句。如果实例对象的属性存在、或者继承自对象的原型，`in` 运算符都会返回

true。比如：

```
var object = {  
    count: 0,  
    related: null  
};  
  
// 好的写法  
if ("count" in object) {  
    // 这里的代码会执行  
}  
  
// 不好的写法：检测假值  
if (object["count"]) {  
    // 这里的代码不会执行  
}  
  
// 好的写法  
if ("related" in object) {  
    // 这里的代码会执行  
}  
  
// 不好的写法：检测是否为 null  
if (object["related"] != null) {  
    // 这里代码不会执行  
}
```

如果你只想检查实例对象的某个属性是否存在，则使用 `hasOwnProperty()`方法。所有继承自 `Object` 的 JavaScript 对象都有这个方法，如果实例中存在这个属性则返回 `true`（如果这个属性只存在于原型里，则返回 `false`）。需要注意的是，在 IE 8 以及更早版本的 IE 中，DOM 对象并非继承自 `Object`，因此也不包含这个方法。也就是说，你在调用 DOM 对象的 `hasOwnProperty()`方法之前应当先检测其是否存在（假如你已经知道对象不是 DOM，则可以省略这一步）。

```
// 对于所有非 DOM 对象来说，这是好的写法  
if (object.hasOwnProperty("related")) {  
    // 执行这里的代码  
}  
  
// 如果你不确定是否为 DOM 对象，则这样来写  
if ("hasOwnProperty" in object && object.hasOwnProperty("related")) {  
    // 执行这里的代码  
}
```

因为存在 IE 8 以及更早版本 IE 的情形，在判断实例对象是否存在时，我更倾向于使用 `in` 运算符，只有在需要判断实例属性时才会用到 `hasOwnProperty()`。

不管你什么时候需要检测属性的存在性，请使用 `in` 运算符或者 `hasOwnProperty()`。这样做可以避免很多 bug。

当然，如果你需要检测值是否为 `null` 或者 `undefined`，则参考第 1 章的内容。

第9章

将配置数据从代码中分离出来

代码无非是定义一些指令的集合让计算机来执行。我们常常将数据传入计算机，由指令对数据进行操作，并最终产生一个结果。当不得不修改数据时问题就来了。任何时候你修改源代码都会有引入 bug 的风险，且只修改一些数据的值也会带来一些不必要的风险，因为数据是不应当影响指令的正常运行的。精心设计的应用应当将关键数据从主要的源码中抽离出来，这样我们修改源码时才更加放心。

9.1 什么是配置数据

配置数据是应用中写死（hardcoded）的值。来看一下这段代码。

```
// 将配置数据埋藏在代码中
function validate(value) {
    if (!value) {
        alert("Invalid value");
        location.href = "/errors/invalid.php";
    }
}

function toggleSelected(element) {
    if (hasClass(element, "selected")) {
        removeClass(element, "selected");
    } else {
        addClass(element, "selected");
    }
}
```

在这段代码中有三个配置数据片段。第一个是字符串“Invalid value”（非法的值），

这个值是用来给用户提示的。因为这个字符串可以被用户接触到，所以它可能会被很频繁地修改。第二个是 URL “/errors /invalid.php”。在开发过程中，当架构变更时则很可能频繁修改 URL。第三个是 CSS 的 className “selected”。有三处都用到了这个 className，这也意味着要想修改这个 className 则必须修改三处，很可能不小心丢掉了某处修改。

我们认为它们都是配置数据，因为它们都是写死在代码里的，且将来可能会被修改。下面给出了一些配置数据的例子。

- URL。
- 需要展现给用户的字符串。
- 重复的值。
- 设置（比如每页的配置项）。
- 任何可能发生变更的值。

我们时刻要记住，配置数据是可发生变更的，而且你不希望因为有人突然想修改页面中展示的信息，而导致你去修改 JavaScript 源码。

9.2 抽离配置数据

将配置数据从代码中抽离出来的第一步是将配置数据拿到外部，即将数据从 JavaScript 代码之中拿掉。下面这个例子即是改造了一下上文的示例代码，将配置数据抽离了出来。

```
// 将配置数据抽离出来
var config = {
    MSG_INVALID_VALUE: "Invalid value",
    URL_INVALID: "/errors/invalid.php",
    CSS_SELECTED: "selected"
};

function validate(value) {
    if (!value) {
        alert(config.MSG_INVALID_VALUE);
        location.href = config.URL_INVALID;
```

```
        }
    }

    function toggleSelected(element) {
        if (hasClass(element, config.CSS_SELECTED)) {
            removeClass(element, config.CSS_SELECTED);
        } else {
            addClass(element, config.CSS_SELECTED);
        }
    }
}
```

在这段代码中，我们将配置数据保存在了 `config` 对象中。`config` 对象的每个属性都保存了一个数据片段，每个属性名都有前缀，用以表明数据的类型（`MSG` 表示展现给用户的消息，`URL` 表示网络地址，`CSS` 表示这是一个 `className`）。当然，命名约定是个人偏好。对于这段代码来说最重要的一点是，所有的配置数据都从函数中移除，并替换为 `config` 对象中的属性占位符。

将配置数据抽离出来意味着任何人都可以修改它们，而不会导致应用逻辑出错。同样，我们可以将整个 `config` 对象放到单独的文件中，这样对配置数据的修改可以完全和使用这些数据的代码隔离开来。

9.3 保存配置数据

配置数据最好放在单独的文件中，以便清晰地分隔数据和应用逻辑。将配置数据放置于单独的 JavaScript 文件中是一个不错的开始。一旦配置数据存放于独立的文件中，我们就可以来管理这些数据。一种值得尝试的做法是将配置数据存放于非 JavaScript 文件中。

尽管你在写一个 JavaScript 应用，但 JavaScript 并非存储数据的最佳方式。因为你必须遵照 JavaScript 语言的语法来组织它们，因此你需要确保不会出语法错误。如果你将多个 JavaScript 文件合并成一个，一个语法错误就会导致整个应用程序崩溃。将文件中的配置数据正确格式化是很麻烦的一件事，当你拿到这样一个文件时，你会觉得将配置数据自动转换为 JavaScript 格式是无足轻重的一件事。

我最喜欢用的一种方式是采用 Java 属性文件（Java properties file）来存放配置数据。Java 属性文件是一组非常简单的名值对，每个名值对独占一行（除非你将它们放于

多行序列中), 形式为 `name=value`。等号两边是否有空格都无所谓, 所以这种语法不容易出错。注释是以#开头的行, 来看下面一段例子。

```
# 面向用户的消息
MSG_INVALID_VALUE = Invalid value

# URLs
URL_INVALID = /errors/invalid.php

# CSS Classes
CSS_SELECTED = selected
```

这个配置文件包含了和上个例子中 `config` 对象一样的属性。可见这种格式是多么简单。而且不用写引号, 也就是说不必担心字符串中的某些转义字符以及字符串的结束位置。同样, 你也不必担心分号和逗号。编辑数据时你可以完全忽略 JavaScript 语法。

接下来的步骤就是将这个文件转换为 JavaScript 可用的文件。有三种常见的格式可供采用。第一种是 JSON, 这是一种很常用的数据格式, 我们常常将 JSON 数据放置于另外一个文件中, 或者向服务器提交查询时也用 JSON 格式。比如:

```
{"MSG_INVALID_VALUE": "Invalid value", "URL_INVALID": "/errors/ invalid.php",
"CSS_SELECTED": "selected"}
```

第二种是 JSONP (JSON with padding), 是将 JSON 结构用一个函数 (调用) 包装起来。

```
myfunc({ "MSG_INVALID_VALUE": "Invalid value", "URL_INVALID": "/errors/
invalid. php",
"CSS_SELECTED": "selected"});
```

因为 JSONP 是一个合法的 JavaScript 文件, 你可以将它和其他的文件合并在一起, 以便其他的逻辑可以读取这段数据。

最后一种是纯 JavaScript, 这种方式是将 JSON 对象赋值给一个变量, 这个变量会被程序用到。比如:

```
var config={"MSG_INVALID_VALUE": "Invalid value", "URL_INVALID": "/errors/
invalid.php",
"CSS_SELECTED": "selected"};
```

和 JSONP 类似，纯 JavaScript 版本可以在上线时很容易地合并入其他的 JavaScript 文件中。

对于这些常见的使用场景来说，我创建了一个工具 Props2Js，它读取 Java 属性文件并给出以上三种格式的输出。Props2Js 是免费且开源的，项目地址在 <https://github.com/nzakas/props2js/>。用法如下。

```
java -jar props2js-0.1.0.jar --to jsonp --name myfunc  
--output result.js source.properties
```

其中--to 选项指定了要输出的格式，可以是“js”、“json”或“jsonp”，--name 选项指定了变量名称（当格式为“js”时）或函数名称（当格式为“jsonp”时）。当格式为“json”时忽略这个选项。--output 选项指定了输出的文件名。因此，这一行命令的意思是读取 Java 属性文件 source.properties 并将结果以 JSONP 格式输出，回调函数名为 myfunc，结果写入文件 result.js 中。

使用类似 Props2Js 之类的工具可以让我们把配置数据放入格式很简单的文件里，并很容易地转换为 JavaScript 格式的文件。

抛出自定义错误

当年，最令我迷惑的是编程语言有“创建”错误的能力。看到 Java 中的 `throw` 操作符，我的第一反应是：“嗯，愚蠢！为什么你要引发一个错误呢？”错误是敌人——是需要竭力避免的——所以“创建”错误的能力在我看来是无益的，是语言的危险面。我曾想，在 JavaScript 中包含那个操作符也是愚蠢的，何况起初人们还不怎么了解这门语言。现在，在积累了大量的经验之后，我已然很热衷于抛出自己的错误。

在 JavaScript 中抛出错误是一门艺术。摸清楚代码中哪里合适抛出错误是需要时间的。因此，一旦搞清楚了这一点，调试代码的时间将大大缩短，对代码的满意度将急剧提升。

10.1 错误的本质

当某些非期望的事情发生时程序就引发一个错误。也许是给一个函数传递了一个不正确的值，或者是数学运算碰到了一个无效的操作数。编程语言定义了一组基本的规则，当偏离了这些规则时将导致错误，然后开发者能修复代码。如果错误没有被抛出或者报告给你的话，调试是非常困难的。如果所有的失败都是悄无声息的，首要的问题是那必将消耗你大量的时间才能发现它，更不要说单独隔离并修复它了。所以，错误是开发者的朋友，而不是敌人。

错误常常在非期望的地点、不恰当的时机跳出来，这很麻烦。更糟糕的是，默认的错误消息通常太简洁而无法解释到底什么东西出错了。JavaScript 错误消息以信息稀少、隐晦含糊而臭名昭著（特别是在老版本的 IE 中），这只会让问题更加复杂化。

想象一下，如果跳出一个错误能这样描述：“由于发生这些情况，该函数调用失败”。那么，调试任务马上就会变得更加简单，这正是抛出自己的错误的好处。

像内置的失败案例一样来考虑错误是非常有帮助的。在代码某个特殊之处计划一个失败总比要在所有的地方都预期失败简单的多。在产品设计上，这是非常普遍的实践经验，而不仅仅是在代码编写方面。汽车尚有碰撞力吸收区域，这些区域框架的设计旨在撞击发生时以可预测的方式崩塌。知道一个碰撞到来时这些框架将如何反应——特别是，哪些部分将失败——制造商将能保证乘客的安全。你的代码也可以用这种方法来创建。

10.2 在 JavaScript 中抛出错误

毫无疑问，在 JavaScript 中抛出错误要比在任何其他语言中做同样的事情更加有价值，这归咎于 Web 端调试的复杂性。可以使用 `throw` 操作符，将提供的一个对象作为错误抛出。任何类型的对象都可以作为错误抛出，然而，`Error` 对象是最常用的。

```
throw new Error("Something bad happened.")
```

内置的 `Error` 类型在所有的 JavaScript 实现中都是有效的，它的构造器只接受一个参数，指代错误消息 (`message`)。当以这种方式抛出错误时，如果没有通过 `try-catch` 语句来捕获的话，浏览器通常直接显示该消息 (`message` 字符串)。当今大多数浏览器都有一个控制台 (`console`)，一旦发生错误都会在这里输出错误信息。换言之，任何你抛出的和没抛出的错误都被以相同的方式来对待。

缺乏经验的开发者有时直接将一个字符串作为错误抛出，如：

```
// 不好的写法
throw "message";
```

这样做确实能够抛出一个错误，但不是所有的浏览器做出的响应都会按照你的预期。Firefox、Opera 和 Chrome 都将显示一条“`uncaught exception`”消息，同时它们包含上述消息字符串。Safari 和 IE 只是简陋地抛出一个“`uncaught exception`”错误，完全不提供上述消息字符串，这种方式对调试无益。

显然，如果愿意，你可以抛出任何类型的数据。没有任何规则约束不能是特定的数据类型。

```
throw { name: "Nicholas" };
throw true;
throw 12345;
throw new Date();
```

就一件事情需要牢记，如果没有通过 try-catch 语句捕获，抛出任何值都将引发一个错误。Firefox、Opera 和 Chrome 都会在该抛出的值上调用 String() 函数，来完成错误消息的显示逻辑，但 Safari 和 IE 不是这样的。针对所有的浏览器，唯一不出差错的显示自定义的错误消息的方式就是用一个 Error 对象。

10.3 抛出错误的好处

抛出自己的错误可以使用确切的文本供浏览器显示。除了行和列的号码，还可以包含任何你需要的有助于调试问题的信息。我推荐总是在错误消息中包含函数名称，以及函数失败的原因。考察下面的函数。

```
function getDivs(element) {
    return element.getElementsByTagName("div");
}
```

这个函数旨在获取 element 元素下所有后代元素中的 (div) 元素。传递给函数要操作的 DOM (元素) 为 null 值可能是件很常见的事情，但实际需要的是 DOM 元素。如果给这个函数传递 null 会发生什么情况呢？你会看到一个类似 “object expected”的含糊的错误消息。然后，你要去看执行栈，再实际定位到源文件中的问题。通过抛出一个错误，调试会更简单。

```
function getDivs(element) {
    if (element && element.getElementsByTagName) {
        return element.getElementsByTagName("div");
    } else {
        throw new Error("getDivs(): Argument must be a DOM element.");
    }
}
```

现在给 getDivs() 函数抛出一个错误，任何时候只要 element 不满足继续执行的条件，就会抛出一个错误明确地陈述发生的问题。如果在浏览器控制台中输出该错误，你

马上能开始调试，并知道最有可能导致该错误的原因是调用函数试图用一个值为 null 的 DOM 元素去做进一步的事情。

我倾向于认为抛出错误就像给自己留下告诉自己为什么失败的便签。

10.4 何时抛出错误

理解了如何抛出错误只是等式的一个部分，另外一部分就是要理解什么时候抛出错误。由于 JavaScript 没有类型和参数检查，大量的开发者错误地假设他们自己应该实现每个函数的类型检查。这种做法并不实际，并且会对脚本的整体性能造成影响。考察下面的函数，它试图实现充分的类型检查。

```
// 不好的写法：检查了太多错误
function addClass(element, className) {
    if (!element || typeof element.className != "string") {
        throw new Error("addClass(): First argument must be a DOM element.");
    }
    if (typeof className != "string") {
        throw new Error("addClass(): Second argument must be a string.");
    }
    element.className += " " + className;
}
```

这个函数本来只是简单地给一个给定的元素增加一个 CSS 类名 (className)，因此，函数的大部分工作变成了错误检查。纵然它能在每个函数中检查每个参数（模仿静态类型语言），在 JavaScript 中这么做也会引起过度的杀伤。辨识代码中哪些部分在特定的情况下最有可能导致失败，并只在那些地方抛出错误才是关键所在。

在上例中，最有可能引发错误的是给函数传递一个 null 引用值。如果第二个参数是 null 或者一个数字或者一个布尔量是不会抛出错误的，因为 JavaScript 会将其强制转换为字符串。那意味着导致 DOM 元素的显示不符合期望，但这并不至于提高到严重错误的程度。所以，我只会检查 DOM 元素。

```
// 好的写法
function addClass(element, className) {
    if (!element || typeof element.className != "string") {
        throw new Error("addClass(): First argument must be a DOM element.");
    }
    element.className += " " + className;
}
```

如果一个函数只被已知的实体调用，错误检查很可能没有必要（这个案例是私有函数）；如果不能提前确定函数会被调用的所有地方，你很可能需要一些错误检查。这就更有可能从抛出自己的错误中获益。抛出错误最佳的地方是在工具函数中，如 `addClass()` 函数，它是通用脚本环境中的一部分，会在很多地方使用。更准确的案例是 JavaScript 类库。

针对已知条件引发的错误，所有的 JavaScript 类库都应该从它们的公共接口里抛出错误。如 jQuery、YUI 和 Dojo 等大型的库，不可能预料你在何时何地调用了它们的函数。当你做错事的时候通知你是它们的责任，因为你不可能进入库代码中去调试错误的原因。函数调用栈应该在进入库代码接口时就终止，不应该更深了。没有比看到由一打库代码中函数调用时发生一个错误更加糟糕的事情了吧，库的开发者应该承担起防止类似情况发生的责任。

私有 JavaScript 库也类似。许多 Web 应用程序都有自己专用的内置的 JavaScript 库或“拿来”一些有名的开源类库（类似 jQuery）。类库提供了对脏的实现细节的抽象，目的是让开发者用得更爽。抛出错误有助于对开发者安全地隐藏这些脏的实现细节。

这里有一些关于抛出错误很好的经验法则。

- 一旦修复了一个很难调试的错误，尝试增加一两个自定义错误。当再次发生错误时，这将有助于更容易地解决问题。
- 如果正在编写代码，思考一下：“我希望[某些事情]不会发生，如果发生，我的代码会一团糟糕”。这时，如果“某些事情”发生，就抛出一个错误。
- 如果正在编写的代码别人（不知道是谁）也会使用，思考一下他们使用的方式，在特定的情况下抛出错误。

请牢记，我们目的不是防止错误，而是在错误发生时能更加容易地调试。

10.5 try-catch 语句

JavaScript 提供了 `try-catch` 语句，它能在浏览器处理抛出的错误之前来解析它。可能引发错误的代码放在 `try` 块中，处理错误的代码放在 `catch` 块中。例如：

```
try {
    somethingThatMightCauseAnError();
} catch (ex) {
    handleError(ex);
}
```

当在 `try` 块中发生了一个错误时，程序立刻停止执行，然后跳到 `catch` 块，并传入一个错误对象。检查该对象可以确定从错误中恢复的最佳动作。

当然，还可以增加一个 `finally` 块。`finally` 块中的代码不管是否有错误发生，最后都会被执行。例如：

```
try {
    somethingThatMightCauseAnError();
} catch (ex) {
    handleError(ex);
} finally {
    continueDoingOtherStuff();
}
```

在某些情况下，`finally` 块工作起来有点复杂（微妙）。例如，如果 `try` 块中包含了一个 `return` 语句，实际上它必须等到 `finally` 块中的代码执行后才能返回。由于这个原因，`finally` 其实不太常用，但如果处理错误必要，它仍然是处理错误的一个强大的工具。

使用 `throw` 还是 `try-catch`

通常，开发者很难敏锐地判断是抛出一个错误还是用 `try-catch` 来捕获一个错误。错误只应该在应用程序栈中最深的部分抛出，如上面的讨论，就像 JavaScript 类库的代码。任何处理应用程序特定逻辑的代码都应该有错误处理的能力，并且捕获从底层组件中抛出的错误。

应用程序逻辑总是知道调用某个特定函数的原因，因此也是最适合处理错误的。千万不要将 `try-catch` 中的 `catch` 块留空，你应该总是写点什么来处理错误。例如，不要像下面这样做。

```
// 不好的写法
try {
    somethingThatMightCauseAnError();
} catch (ex) {
```

```
// 空  
}
```

如果知道可能要发生错误，那肯定知道如何从错误中恢复。确切地说，如何从错误中恢复在开发模式中与实际放到生产环节中是不一样的，这没关系。最重要的是，你实实在在地在处理错误，而不是忽略它。

10.6 错误类型

ECMA-262 规范指出了 7 种错误类型。当不同错误条件发生时，这些类型在 JavaScript 引擎中都有用到，当然我们也可以手动创建它们。

Error

所有错误的基本类型。实际上引擎从来不会抛出该类型的错误。

Evaluator

通过 eval() 函数执行代码发生错误时抛出。

RangeError

一个数字超出它的边界时抛出——例如，试图创建一个长度为 -20 的数组（new Array(-20)）。该错误在正常的代码执行中非常罕见。

ReferenceError

期望的对象不存在时抛出——例如，试图在一个 null 对象引用上调用一个函数。

SyntaxError

给 eval() 函数传递的代码中有语法错误时抛出。

TypeError

变量不是期望的类型时抛出。例如，new 10 或 “prop” in true。

URIError

给 encodeURI()、encodeURIComponent()、decodeURI() 或者 decodeURIComponent() 等函数传递格式非法的 URI 字符串时抛出。

理解错误的不同类型可以帮助我们更容易地处理它。所有的错误类型都继承自 `Error`，所以用 `instanceof Error` 检查其类型得不到任何有用的信息。通过检查特定的错误类型可以更可靠地处理错误。

```
try {
    // 有些代码引发了错误
} catch (ex) {
    if (ex instanceof TypeError) {
        // 处理 TypeError 错误
    } else if (ex instanceof ReferenceError) {
        // 处理 ReferenceError 错误
    } else {
        // 其他处理
    }
}
```

如果抛出自己的错误，并且是数据类型而不是一个错误，你可以非常轻松地区分自己的错误和浏览器的错误类型的不同。但是，抛出实际类型的错误与抛出其他类型的对象相比，有几大优点。

首先，如上讨论，在浏览器正常错误处理机制中会显示错误消息。其次，浏览器给抛出的 `Error` 对象附加了一些额外的信息。这些信息不同浏览器各不相同，但它们为错误提供了如行、列号等上下文信息，在有些浏览器中也提供了堆栈和源代码信息。当然，如果用了 `Error` 的构造器，你就丧失了区分自己抛出的错误和浏览器错误的能力。

解决方案就是创建自己的错误类型，让它继承自 `Error`。这种做法允许你提供额外的信息，同时可区别于浏览器抛出的错误。可以用如下的模式来创建自定义的错误类型。

```
function MyError(message) {
    this.message = message;
}
MyError.prototype = new Error();
```

这段代码有两个重要的部分：`message` 属性，浏览器必须要知道的错误消息字符串；设置 `prototype` 为 `Error` 的一个实例，这样对 JavaScript 引擎而言就标识它是一个错误对象了。接下来就可以抛出一个 `MyError` 的实例对象，使得浏览器能像处理原生错误一样做出响应。

```
throw new MyError("Hello world!");
```

提醒一下，该方法在 IE 8 和更早的浏览器中不显示错误消息。相反，会看见那个通用的“Exception thrown but not caught”消息。这个方法最大的好处是，自定义错误类型可以检测自己的错误。

```
try {  
    // 有些代码引发了错误  
} catch (ex) {  
    if (ex instanceof MyError) {  
        // 处理自己的错误  
    } else {  
        // 其他处理  
    }  
}
```

如果总是捕获你自己抛出的所有错误，那么 IE 的那点儿小愚蠢也不足为道了。在一个正确的错误处理系统中获得的好处是巨大的。该方法可以给出更多、更灵活的信息，告知开发者如何正确地处理错误。

第 11 章

不是你的对象不要动

JavaScript 独一无二之处在于任何东西都不是神圣不可侵犯的。默认情况下，你可以修改任何你可以触及的对象。它（解析器）根本就不在乎这些对象是开发者定义的还是默认执行环境的一部分——只要是能访问到的对象都可以修改。在一个开发者独自开发的项目中，这不是问题，开发者确切地知道正在修改什么，因为他对所有代码都了如指掌。然而，在一个多人开发的项目中，对象的随意修改就是个大问题了。

11.1 什么是你的

当你的代码创建了这些对象时，你拥有这些对象。创建了对象的代码也许没必要一定由你来编写，但只要维护代码是你的责任，那么就是你拥有这些对象。举例来说，YUI 团队拥有该 YUI 对象，Dojo 团队拥有该 dojo 对象。即使编写代码定义该对象的原始作者离开了，各自对应的团队仍然是这些对象的拥有者。

当在项目中使用一个 JavaScript 类库，你个人不会自动变成这些对象的拥有者。在一个多人开发的项目中，每个人都假设库对象会按照它们的文档中描述的一样正常工作。如果你在使用 YUI，修改了其中的对象，那么这就给你自己的团队设置了一个陷阱。这必将导致一些问题，有些人可能会掉进去。

请牢记，如果你的代码没有创建这些对象，不要修改它们，包括：

- 原生对象（Object、Array 等等）。

- DOM 对象（例如，`document`）。
- 浏览器对象模型（BOM）对象（例如，`window`）。
- 类库的对象。

上面所有这些对象是你项目执行环境的一部分。由于它们已经存在了，你可以直接使用这些或者用其来构建某些新的功能，而不应该去修改它们。

11.2 原则

企业软件需要一致而可靠的执行环境使其方便维护。在其他语言中，考虑将已存在的对象作为库用来完成开发任务。在 JavaScript 中，我们可以将已存在的对象视为一种背景，在这之上可以做任何事情。你应该把已存在的 JavaScript 对象如一个实用工具函数库一样来对待。

- 不覆盖方法。
- 不新增方法。
- 不删除方法。

当项目中只有你一个开发者时，因为你了解它们，对它们有预期，这些种类的修改很容易处理。当与一个团队一起在做一个大型的项目时，像这些情况的修改会导致大量的混乱，也会浪费很多时间。

11.2.1 不覆盖方法

在 JavaScript 中，有史以来最糟糕的实践是覆盖一个非自己拥有的对象的方法。当我在 My Yahoo! 团队中工作时就因此而导致了很多问题。遗憾的是，JavaScript 中覆盖一个已存在的方法是难以置信的容易。即使那个神圣的 `document.getElementById()` 方法也不例外，可以被轻而易举地覆盖。

```
// 不好的写法
document.getElementById = function() {
    return null;           // 引起混乱
};
```

如上例所示，没有任何方法能阻止覆盖 DOM 方法。更严重的是，页面中所有脚本都可以覆盖其他脚本的方法。所以任何脚本都可以覆盖 `document.getElementById()` 方法使它返回 `null`，这会让 JavaScript 库和其他依赖于该方法的代码都失效。那会完全丧失原始的功能，而再也无法恢复。

也许，你看到过类似下面这样的模式。

```
// 不好的写法
document._originalGetElementById = document.getElementById;
document.getElementById = function(id) {
    if (id == "window") {
        return window;
    } else {
        return document._originalGetElementById(id);
    }
};
```

上例中，将一个原生方法 `document.getElementById()` 的“指针”保存在 `document._originalGetElementById()` 中，以便后续使用。然后，`document.getElementById()` 被一个新的方法覆盖了。新方法有时也会调用原始的方法，其中有一种情况不调用。这种“覆盖加可靠退化”的模式至少和原生方法一样不好，也许会更糟，因为 `document.getElementById()` 时而符合预期，时而不符合。^①

我亲身经历并处理过有人覆盖一个已存在对象的方法所带来的副作用。当时，我在 My Yahoo! 团队工作，由于有人覆盖了 YUI2 的 `YAHOO.util.Event.stopEvent()` 方法来做一些其他事情。跟踪这个问题花费了好几天，因为我们都假设该方法只做了它以前应该做的事情，所以我们用 `debugger` 打断点时从来没有跟踪到该方法的内部。直到我们发现这个方法原来被重写了，同时也发现更多其他的 bug，因为同一个方法的原始用法在其他的页面中也用到了——但那时这个方法的行为并不像预想的那样。解决这个问题是难以置信的麻烦，在一个大型的项目中，一个此类问题就会导致浪费大量时间和金钱。

11.2.2 不新增方法

在 JavaScript 中为已存在的对象新增方法是很简单的。只需要创建一个函数赋值给一个已存在的对象的属性，使其成为方法即可。这种做法可以修改所有类型的对象。

^① 译注：这种做法亦被称为“函数劫持”。

```
// 不好的写法 - 在 DOM 对象上增加了方法
document.sayImAwesome = function() {
    alert("You're awesome.");
};

// 不好的写法 - 在原生对象上增加了方法
Array.prototype.reverseSort = function() {
    return this.sort().reverse();
};

// 不好的写法 - 在库对象上增加了方法
YUI.doSomething = function() {
    // 代码
};


```

几乎不可能阻止你为任何对象添加方法。为非自己拥有的对象增加方法一个大问题，会导致命名冲突。因为一个对象此刻没有某个方法不代表它未来也没有。更糟糕的是如果将来原生的方法和你的方法行为不一致，你将陷入一场代码维护的噩梦。

我们要从 Prototype JavaScript 类库的发展历史中吸取教训。从修改各种各样的 JavaScript 对象角度而言 Prototype 非常著名。它很随意地为 DOM 和原生的对象增加方法。实际上，库的大多数代码定义为扩展已存在的对象，而不是自己创建对象。Prototype 的开发者将该库看作是对 JavaScript 的补充。在小于 1.6 的版本中，Prototype 实现了一个 `document.getElementsByClassName()` 方法。也许你认识该方法，因为在 HTML5 中是官方定义的，它标准化了 Prototype 的用法。

Prototype 的 `document.getElementsByClassName()` 方法返回包含了指定 CSS 类名的元素的一个数组。Prototype 在数组上也增加了一个方法，`Array.prototype.each()`，它在该数组上迭代并在每个元素上执行一个函数。这让开发者可以编写如下代码。

```
document.getElementByClassName("selected").each(doSomething);
```

在 HTML5 标准化该方法和浏览器开始原生地实现之前，代码是没有问题的。当 Prototype 团队知道原生的 `document.getElementsByClassName()` 即将到来，所以他们增加了一些防守性的代码，如下。

```
if (!document.getElementsByClassName) {
    document.getElementsByClassName = function(classes) {
        // 非原生实现
    };
}
```

故，Prototype 只是在 `document.getElementsByClassName()` 不存在的时候定义它。这看上去好像问题就此解决了，但还有一个重要的事实是：HTML5 的 `document.getElementsByClassName()` 不返回一个数组，所以 `each()` 方法根本不存在。原生的 DOM 方法使用了一个特殊化的集合类型称为 `NodeList`。`document.getElementsByClassName()` 返回一个 `NodeList` 来匹配其他的 DOM 方法的调用。

如果浏览器中原生实现了 `document.getElementsByClassName()` 方法，那么由于 `NodeList` 没有 `each()` 方法，无论是原生的还是 Prototype 增加的 `each()` 方法，在执行时都将引发一个 JavaScript 错误。最后的结局是 Prototype 的用户不得不既要升级类库代码还要修改他们自己的代码，真是一场维护的噩梦。

从 Prototype 的错误中可以学到，你不可能精确预测 JavaScript 将来会如何变化。标准已经进化了，它们经常会从诸如 Prototype 这样的库代码中获得一些线索来决定下一代标准的新功能。事实上，原生的 `Array.prototype.forEach()` 方法在 ECMAScript 5 有定义，它与 Prototype 的 `each()` 方法行为非常类似。问题是不知道官方的功能与原先会有怎么样的不同，甚至是微小的区别也将导致很大的问题。

大多数 JavaScript 库代码有一个插件机制，允许为代码库安全地新增一些功能。如果想修改，最佳最可维护的方式是创建一个插件。

11.2.3 不删除方法

删除 JavaScript 方法和新增方法一样简单。当然，覆盖一个方法也是删除已存在的方法的一种方式。最简单的删除一个方法的方式就是给对应的名字赋值为 `null`。

```
// 不好的写法 - 删除了 DOM 方法
document.getElementById = null;
```

将一个方法设置为 `null`，不管它以前是怎么定义的，现在它已经不能被调用到了。如果方法是在对象的实例上定义的（相对于对象的原型而言），也可以使用 `delete` 操作符来删除。

```
var person = {
    name: "Nicholas"
};
delete person.name;
console.log(person.name);      // 未定义
```

上例中，从 person 对象中删除了 name 属性。`delete` 操作符只能对实例的属性和方法起作用。如果在 `prototype` 的属性或方法上使用 `delete` 是不起作用的。例如：

```
// 不影响  
delete document.getElementById;  
console.log(document.getElementById("myelement")); // 仍然能工作
```

因为 `document.getElementById()` 是原型上的一个方法，使用 `delete` 是无法删除的。但是，仍然可以用对其赋值为 `null` 的方式来阻止被调用。

无需赘述，删除一个已存在对象的方法是糟糕的实践。不仅有依赖那个方法的开发者存在，而且使用该方法的代码有可能已经存在了。删除一个在用的方法会导致运行时错误。如果你的团队不应该使用某个方法，将其标识为“废弃”，可以用文档或者用静态代码分析器。删除一个方法绝对应该是最后的选择。

反之，不删除你拥有对象的方法实际上是比较好的实践。从库代码或原生对象上删除方法是非常难的事情，因为第三方代码正依赖于这些功能。在很多案例中，库代码和浏览器都会将有 `bug` 或不完整的方法保留很长一段时间，因为删除它们以后会在数不胜数的网站上导致错误。

11.3 更好的途径

修改非自己拥有的对象是解决某些问题很好的方案。在一种“无公害”的状态下，它通常不会发生；发生的原因可能是开发者遇到了一个问题，然而又通过修改对象解决了这个问题。尽管如此，解决一个已知问题的方案总是不止一种的。大多是计算机科学知识已经在静态类型语言环境中进化出了解决难题方案，如 Java。可能有一些方法，所谓的设计模式，不直接修改这些对象而是扩展这些对象。

在 JavaScript 之外，最受欢迎的对象扩充的形式是继承。如果一种类型的对象已经做到了你想要的大多数工作，那么继承自它，然后再新增一些功能即可。在 JavaScript 中有两种基本的形式：基于对象的继承和基于类型的继承。

在 JavaScript 中，继承仍然有一些很大的限制。首先，（还）不能从 DOM 或 BOM 对象继承。其次，由于数组索引和 `length` 属性之间错综复杂的关系，继承自 `Array` 是不能正常工作的。

11.3.1 基于对象的继承

在基于对象的继承中，也经常叫作原型继承，一个对象继承另外一个对象是不需要调用构造函数的。ECMAScript 5 的 `Object.create()`方法是实现这种继承的最简单的方式。例如：

```
var person = {
    name: "Nicholas",
    sayName: function() {
        alert(this.name);
    }
};

var myPerson = Object.create(person);
myPerson.sayName(); // 弹出“Nicholas”
```

这个例子创建了一个新对象 `myPerson`，它继承自 `person`。这种继承方式就如同 `myPerson` 的原型设置为 `person`，从此 `myPerson` 可以访问 `person` 的属性和方法，而不需要同名变量在新的对象上再重新定义一遍。例如，重新定义 `myPerson.sayName()` 会自动切断对 `person.sayName()` 的访问。

```
myPerson.sayName = function() {
    alert("Anonymous");
};

myPerson.sayName(); // 弹出“Anonymous”
person.sayName(); // 弹出“Nicholas”
```

`Object.create()` 方法可以指定第二个参数，该参数对象中的属性和方法将添加到新的对象中。例如：

```
var myPerson = Object.create(person, {
    name: {
        value: "Greg"
    }
});

myPerson.sayName(); // 弹出“Greg”
person.sayName(); // 弹出“Nicholas”
```

这个例子创建的 `myPerson` 对象拥有自己的 `name` 属性值，所以调用 `sayName()` 显示的是“`Greg`”而不是“`Nicholas`”。

一旦以这种方式创建了一个新对象，该新对象完全可以随意修改。毕竟，你是该对象的拥有者，在自己的项目中你可以任意新增方法，覆盖已存在方法，甚至是删除方法（或者阻止它们的访问）。

11.3.2 基于类型的继承

基于类型的继承和基于对象的继承工作方式是差不多的，它从一个已存在的对象继承，这里的继承是依赖于原型的。因此，基于类型的继承是通过构造函数实现的，而非对象。这意味着，需要访问被继承对象的构造函数。本书中前面有个例子是基于类型的继承。

```
function MyError(message) {  
    this.message = message;  
}  
  
MyError.prototype = new Error();
```

在上例中，`MyError` 类继承自 `Error`（所谓的超类）。给 `MyError.prototype` 赋值为一个 `Error` 的实例。然后，每个 `MyError` 实例从 `Error` 那里继承了它的属性和方法，`instanceof` 也能正常工作。

```
var error = new MyError("Something bad happened.");  
  
console.log(error instanceof Error);      // true  
console.log(error instanceof MyError);    // true
```

比起 JavaScript 中原生的类型，在开发者定义了构造函数的情况下，基于类型的继承是最合适的。同时，基于类型的继承一般需要两步：首先，原型继承；然后，构造器继承。构造器继承是调用超类的构造函数时传入新建的对象作为其 `this` 的值。例如：

```
function Person(name) {  
    this.name;  
}  
  
function Author(name) {  
    Person.call(this, name);    // 继承构造器  
}  
  
Author.prototype = new Person();
```

这段代码里，Author 类型继承自 Person。属性 name 实际上是由 Person 类管理的，所以 Person.call(this, name) 允许 Person 构造器继续定义该属性。Person 构造器是在 this 上执行的，this 指向一个 Author 对象，所以最终的 name 定义在这个 Author 对象上。

对比基于对象的继承，基于类型的继承在创建新对象时更加灵活。定义了一个类型可以让你创建多个实例对象，所有的对象都是继承自一个通用的超类。新的类型应该明确定义需要使用的属性和方法，它们与超类中的应该完全不同。

11.3.3 门面模式

门面模式是一种流行的设计模式，它为一个已存在的对象创建一个新的接口。门面是一个全新的对象，其背后有一个已存在的对象在工作。门面有时也叫包装器，它们用不同的接口来包装已存在的对象。你的用例中如果继承无法满足要求，那么下一步骤就应该创建一个门面，这比较合乎逻辑。

jQuery 和 YUI 的 DOM 接口都使用了门面。如上所述，你无法从 DOM 对象上继承，所以唯一的能够安全地为其新增功能的选择就是创建一个门面。下面是一个 DOM 对象包装器代码示例。

```
function DOMWrapper(element) {
    this.element = element;
}

DOMWrapper.prototype.addClass = function(className) {
    element.className += " " + className;
};

DOMWrapper.prototype.remove = function() {
    this.element.parentNode.removeChild(this.element);
};

// 用法
var wrapper = new DOMWrapper(document.getElementById("my-div"));
// 添加一个 className
wrapper.addClass("selected");
// 删除元素
wrapper.remove();
```

DOMWrapper 类型期望传递给其构造器的是一个 DOM 元素。该元素会保存起来以便以后引用，它还定义了一些操作该元素的方法。addClass()方法是为那些还未实现 HTML5 的 classList 属性的元素增加 className 的一个简单的方法。remove()

方法封装了从 DOM 中删除一个元素的操作，屏蔽了开发者要访问该元素父节点的需求。

从 JavaScript 的可维护性而言，门面是非常合适的方式，自己可以完全控制这些接口。你可以允许访问任何底层对象的属性或方法，反之亦然，也就是有效地过滤对该对象的访问。你也可以对已有的方法进行改造，使其更加简单易用（上段示例代码就是一个案例）。底层的对象无论如何改变，只要修改门面，应用程序就能继续正常工作。

门面实现一个特定接口，让一个对象看上去像另一个对象，就称作一个适配器。门面和适配器唯一的不同是前者创建新接口，后者实现已存在的接口。

11.4 关于 Polyfill 的注解

随着 ECMAScript 5 和 HTML5 的特性开始在各种浏览器中的实现，JavaScript polyfills（也称为 shims）变得流行起来了。一个 polyfill 是指一种功能的模拟，这些功能在新版本的浏览器中已经有完备定义并原生实现了。例如，ECMAScript 5 为数组增加了 `forEach()` 函数。该方法可以在 ECMAScript 3 中模拟，以便在老版本的浏览器中如同新版本一样使用。polyfills 的关键是它们模拟的原生功能要以完全兼容的方式来实现。因此在有些浏览器中存在了这些功能，所以有必要检测不同情况下它们的处理是否符合标准的方式。

为了达到目的，polyfills 经常会给非自己拥有的对象新增一些方法。我不是 polyfills 的粉丝，不过对于别人使用它们，我表示理解。相比其他的对象修改而言，polyfills 是有界限的，是相对安全的。因为原生实现中是存在这些方法并能工作的，有且仅当原生方法不存在时，polyfills 才新增这些方法，并且它们和原生版本方法的行为是完全一致的。

polyfills 的优点是，当只支持浏览器的原生功能时，它们非常容易删除。如果你选择使用某个 polyfill，你自己做好严格审查。要保证它的功能和原生的版本尽可能的近似，多检查一下这种库代码有单元测试并严格验证了这些功能。polyfills 的缺点是，它们可能没有精确地实现它们（原生浏览器环境）所缺失的功能，从而给你带来的麻烦比缺失的功能要多得多。

从最佳的可维护性角度而言，避免使用 polyfills，相反可以在已存在的功能之上创建门面来实现。这种方法给了你最大的灵活性，当原生实现中有 bug 时这种做法（避免使用 polyfills）就显得特别重要。这种情况下，你根本不想直接使用原生的 API，不然无法将原生实现带有的 bug 隔离开来。

11.5 阻止修改

ECMAScript 5 引入了几个方法来防止对对象的修改。理解这些能力很重要，因此现在可以做到这样的事情：锁定这些对象，保证任何人不能有意或无意地修改他们不想要的功能。在 IE 9、Firefox 4+、Safari 5.1+、Opera 12+ 和 Chrome 中已经提供这些功能。有三种锁定修改的级别。

防止扩展

禁止为对象“添加”属性和方法，但已存在的属性和方法是可以被修改或删除。

密封

类似“防止扩展”，而且禁止为对象“删除”已存在的属性和方法。

冻结

类似“密封”，而且禁止为对象“修改”已存在的属性和方法（所有字段均只读）。

每种锁定的类型都拥有两个方法：一个用来实施操作，另一个用来检测是否应用了相应的操作。如防止扩展，`Object.preventExtension()` 和 `Object.isExtensible()` 两个函数可以使用。

```
var person = {  
    name: "Nicholas"  
};  
  
// 锁定对象  
Object.preventExtension(person);  
console.log(Object.isExtensible(person));      // false  
person.age = 25;      // 正常情况悄悄地失败，除非在 strict 模式下抛出错误
```

在上面的例子中，锁定了 `person` 对象防止被扩展，所以调用 `Object.isExtensible()`

函数返回 `false`。在非严格模式下，试图为 `person` 对象新增属性或方法将会悄无声息地失败。在严格模式下，试图为一个不可扩展的对象新增任何属性或方法将会抛出一个错误。

使用 `Object.seal()` 函数来密封一个对象。可以使用 `Object.isSealed()` 函数来检测一个对象是否已被密封。

```
// 锁定对象
Object.seal(person);
console.log(Object.isExtensible(person)); // false
console.log(Object.isSealed(person)); // true
delete person.name; // 正常情况悄悄地失败，除非在 strict 模式下抛出错误
person.age = 25; // 同上
```

当一个对象被密封时，它已存在的属性和方法不能被删除，故在非严格模式下，试图删除上例中 `name` 属性将会悄悄地失败。在严格模式下，试图删除属性或方法将会抛出一个错误。被密封的对象同时也是不可扩展的，所以调用 `Object.isExtensible()` 函数返回 `false`。

使用 `Object.freeze()` 函数来冻结一个对象。可以使用 `Object.isFrozen()` 函数来检查一个对象是否已被冻结。

```
// 锁定对象
Object.freeze(person);
console.log(Object.isExtensible(person)); // false
console.log(Object.isSealed(person)); // true
console.log(Object.isFrozen(person)); // true
person.name = "Greg"; // 正常情况悄悄地失败，除非在 strict 模式下抛出错误
person.age = 25; // 同上
delete person.name; // 同上
```

被冻结的对象同时也是不可扩展和被密封的，所以调用 `Object.isExtensible()` 函数返回 `false`，调用 `Object.isSealed()` 函数返回 `true`。被冻结的对象和被密封的对象最大的区别在于，前者禁止任何对已存在属性和方法的修改。在非严格模式下，任何这种操作都将会悄悄地失败，在严格模式下都将会抛出一个错误。

使用 ECMAScript 5 中的这些方法是保证你的项目不经过你同意锁定修改的极佳的做法。如果你是一个代码库的作者，很可能想锁定核心库某些部分来保证它们不被意外修改，或者想强迫允许扩展的地方继续存活。如果你是一个应用程序的开发者，锁定应用程序的任何不想被修改的部分。这两种情况中，在全部定义好这些对

象的功能之后，才能使用上述的锁定方法。一旦一个对象被锁定了，它将无法解锁。

如果决定将你的对象锁定修改，我强烈推荐使用严格模式。在非严格模式下，试图修改不可修改的对象总是悄无声息地失败，这在调试期间非常令人沮丧。通过使用严格模式，同样的尝试将抛出一个错误，使得不能修改的原因更加明显。

将来，原生 JavaScript 对象和 DOM 对象很有可能都将统一内置使用 ECMAScript 5 的锁定修改的保护功能。



浏览器嗅探

浏览器嗅探在 Web 开发领域始终是一个热点话题。自从网景浏览器介入以后，这场争论就开始了。几年后，JavaScript 的浏览器嗅探技术也应运而生，网景浏览器是第一个真正流行和被广泛使用的网络浏览器。网景 2.0 远远超过了任何其他在用的 Web 浏览器，以至于网站在返回有用的内容之前都会先查找其特定的用户代理（User Agent）字符串。这迫使其他浏览器厂商，尤其是微软，在他们的用户代理字符串中也包含了这样的信息，以此来绕过这种形式的浏览器嗅探。

12.1 User-Agent 检测

最早的浏览器嗅探即用户代理（user-agent）检测，服务端（以及后来的客户端）根据 user-agent 字符串来确定浏览器的类型。在此期间，服务器会完全根据 user-agent 字符串屏蔽某些特定的浏览器查看网站内容。其中获益最大的浏览器就是网景浏览器。不可否认，网景（在当时）是最强大的浏览器，以至于很多网站都认为只有网景浏览器才会正常展现他们的网页。网景浏览器的 user-agent 字符串如下。

```
Mozilla/2.0 (Win95; I)
```

当 Internet Explorer 首次发布，基本上就被迫沿用了网景浏览器 user-agent 字符串的很大一部分，以此确保服务器能够为这款新的浏览器提供服务。因为绝大多数的用户代理检测的过程都是查找"Mozilla"字符串和斜线之后的版本号，Internet Explorer 浏览器的 user-agent 字符串如下。

Internet Explorer 采用了这样的用户代理字符串，这意味着每个浏览器类型检测也会把这款新的浏览器识别为网景的 Navigator 浏览器。这也使得新生浏览器部分复制现有浏览器用户代理字符串成为了一种趋势。Chrome 发行版的用户代理字符串包含了 Safari 的一部分，而 Safari 的用户代理字符串又相应包含了 Firefox 的一部分，Firefox 又依次包含了 Netscape（网景）用户代理字符串的一部分。

快进到 2005 年，JavaScript 开始越来越流行。服务器可以获取到的浏览器的 user-agent 字符串，在客户端通过 JavaScript 的 navigator.userAgent 同样可以获取。用户代理字符串检测开始转移到 Web 页面中，通过 JavaScript 对 user-agent 字符串执行与服务端相同类型的检测，诸如：

```
// 不好的做法
if (navigator.userAgent.indexOf("MSIE") > -1) {
    // 是 Internet Explorer
} else {
    // 不是 Internet Explorer
}
```

随着越来越多的网站通过 JavaScript 检测用户代理，一批新的网站开始在浏览器中无法正常浏览。近十年前服务器曾遭遇过同样的问题，这个问题以 JavaScript 的形式重演了。

最大的问题是，解析 user-agent 字符串并非易事，由于浏览器为了确保其兼容性，都会复制另一个浏览器的用户代理字符串。因此随着每一个新浏览器的出现，用于用户代理检测的代码都需要更新，于是从新的浏览器发布，到代码更新并部署的这段时间便意味着不计其数的人会遭遇到糟糕的用户体验。

但这也并不是说没有一个有效的方法合理使用 user-agent 字符串。无论是 JavaScript 还是服务器，都有精心编写的库可以提供相当不错的检测机制。遗憾的是，这些库也需要不断更新以适应浏览器的发展包括浏览器新的发行版。整个方法长期而言并不具备很好的可维护性。

为了保证 JavaScript 的正确执行，用户代理检测应该是没有办法的办法。但若你选择使用用户代理检测，最安全的方法是只检测旧的浏览器。例如，假如你需要针对 Internet Explorer 8 和之前的版本执行一些特殊的操作以确保代码能够正常工作，你

应当只检测 Internet Explorer 8 和之前的版本而不要试图检测 Internet Explorer 9 和更高版本，诸如：

```
if (isInternetExplorer8OrEarlier) {  
    // 处理 IE 8 以及更早版本  
} else {  
    // 处理其他浏览器  
}
```

这么做的好处是，Internet Explorer 8 和早期版本的 user-agent 字符串是众所周知且不会改变的。即使你的代码持续运行直至 Internet Explorer 25 发布，代码也很可能会继续正常工作而不需要额外的修改。相反，如果你尝试检测 Internet Explorer 9 和更高版本，你将需要持续不断地更新代码。

浏览器并不总是提供它们原始的 user-agent 字符串。几乎所有的浏览器都可以使用现成的工具来修改用户代理。开发者往往担心这一点会导致用户代理检测失效，即使有时这么做可能是没有办法的办法，因为“你永远不能肯定（这样做是否奏效）”。我的建议是不用担心用户代理欺骗。因为如果一个用户知道如何切换他的用户代理字符串，那么他也肯定能够意识到这么做会导致网站的访问有不可预期后果。如果浏览器标识自身是 Firefox 但是行为却不像 Firefox，那不是你的错。试图去猜测获取到的用户代理字符串是毫无意义的。

12.2 特性检测

我们希望有一种更聪明的基于浏览器条件（进行检测）的方法，于是开发人员转向了一种叫做特性检测的技术。特性检测的原理是为特定浏览器的特性进行测试，并仅当特性存在时即可应用特性检测。因此不要这么做：

```
// 不好的写法  
if (navigator.userAgent.indexOf("MSIE 7") > -1) {  
    // 做些什么  
}  
你应当这么做：  
// 好的写法  
if (document.getElementById) {  
    // 做些什么  
}
```

这两种方法之间是有区别的。前者根据名称和版本对特定浏览器做探测；后者对特

定的功能进行探测，即 `document.getElementById`。因此通过 `user-agent` 嗅探的结果可以知道确切的浏览器和版本号（或者至少是浏览器自己提供的），而特性检测则是判断给定对象或者方法是否存在。注意，这两个方法的结果是截然不同的。

因为特性检测不依赖于所使用的浏览器，而仅仅依据特性是否存在，所以并不一定需要新浏览器的支持。例如，在 DOM 早期的时候，并非所有浏览器都支持 `document.getElementById()`，所以根据 ID 获取元素的代码看起来就有些冗余。

```
// 好的写法
function getById (id) {
    var element = null;

    if (document.getElementById) { // DOM
        element = document.getElementById(id);
    } else if (document.all) { // IE
        element = document.all[id];
    } else if (document.layers) { // Netscape <= 4
        element = document.layers[id];
    }

    return element;
}
```

这么使用特性检测是正确和恰当的，因为代码对特性做检测，当特性存在时才使用。首先检测 `document.getElementById` 是因为它是基于标准的方法。其后是两个特定浏览器的方法。如果这些特性都不存在，该方法会返回 `null`。该函数最大的好处就是当 Internet Explorer 5 和 Netscape 6 发布并支持 `document.getElementById()` 的时候，代码是无需做任何改动的。

先前的例子说明了正确的特性检测的一些重要组成部分。

1. 探测标准的方法。
2. 探测不同浏览器的特定方法。
3. 当被探测的方法均不存在时提供一个合乎逻辑的备用方法。

这种方法同样适用于当今最新的（浏览器）特性检测，浏览器已经实验性地实现了这些最新的特性，而规范还正在最后确定中。例如，`requestAnimationFrame()` 方法

直至 2011 年末才被确定，在这段时间里，几个浏览器已经通过加厂商前缀的方式实现了它们自己的版本。合理的 requestAnimationFrame() 特性检测代码如下。

```
// 好的写法
function setAnimation (callback) {

    if (window.requestAnimationFrame) { // 标准
        return requestAnimationFrame(callback);
    } else if (window.mozRequestAnimationFrame) { // Firefox
        return mozRequestAnimationFrame(callback);
    } else if (window.webkitRequestAnimationFrame) { // Webkit
        return webkitRequestAnimationFrame(callback);
    } else if (window.oRequestAnimationFrame) { // Opera
        return oRequestAnimationFrame(callback);
    } else if (window.msRequestAnimationFrame) { // IE
        return msRequestAnimationFrame(callback);
    } else {
        return setTimeout(callback, 0);
    }

}
```

该代码首先寻找标准的 requestAnimationFrame 方法，并且仅当标准方法不存在时才会继续寻找不同浏览器各自的实现。最后当这些方法浏览器都不支持的时候，会选择用 setTimeout() 代替。再者，即便浏览器更换为标准的实现，这段代码也不需要更新。

12.3 避免特性推断

一种不当的使用特性检测的情况是“特性推断”(Feature Inference)。特性推断尝试使用多个特性但仅验证了其中之一。根据一个特性的存在推断另一个特性是否存在。问题是，推断是假设并非事实，而且可能会导致维护性的问题。例如，如下是一些使用特性推断的旧代码。

```
// 不好的写法 - 使用特性推断
function getById (id) {

    var element = null;

    if (document.getElementsByTagName) { // DOM
        element = document.getElementById(id);
    } else if (window.ActiveXObject) { // IE
```

```

        element = document.all[id];
    } else { // Netscape <= 4
        element = document.layers[id];
    }

    return element;
}

```

该函数是最糟糕的特性推断，其中做出了如下几个推断。

- 如果 `document.getElementsByTagName()` 存在，则 `document.getElementById()` 也存在。实际上，这个假设是从一个 DOM 方法的存在推断出所有方法都存在。
- 如果 `window.ActiveXObject` 存在，则 `document.all` 也存在。这个推断基本上断定 `window.ActiveXObject` 仅仅存在于 Internet Explorer，且 `document.all` 也仅存在于 Internet Explorer，所以如果你判断一个存在，其他的也必定存在。实际上，Opera 的一些版本也支持 `document.all`。
- 如果这些推断都不成立，则一定是 Netscape Navigator 4 或者更早的版本。这看似正确，但极其不严格。

你不能从一个特性的存在推断出另一个特性是否存在。最好的情况下两者有薄弱的联系，最坏的情况下两者根本没有直接关系。也就好比说是，“如果它看起来一个鸭子，就必定像鸭子一样嘎嘎地叫。”

12.4 避免浏览器推断

在某些时候，用户代理检测和特性检测让许多 Web 开发人员很困惑。于是写出来的代码就变成了这样。

```

// 不好的写法
if (document.all) { // IE
    id = document.uniqueID;
} else {
    id = Math.random();
}

```

这段代码的问题是，通过检测 `document.all`，间接地判断浏览器是否为 Internet Explorer。一旦确定了浏览器是 Internet Explorer，便假设可以安全地使用 IE 所特有

的 `document.uniqueID`。然而，你所做的所有探测仅仅说明 `document.all` 是否存在，而并不能用于判断浏览器是否是 Internet Explorer。正因为 `document.all` 的存在并不意味着 `document.uniqueID` 也是可用的，因此这是一个错误的隐式推断，可能会导致代码不能正常运行。

为了更清楚地表述该问题，开发人员开始将这种代码。

```
var isIE = navigator.userAgent.indexOf("MSIE") > -1;
```

修改为如下这样。

```
// 不好的写法  
var isIE = !!document.all;
```

这种转变体现了一种对“不要使用用户代理检测”的误解。虽然不是直接检测特定的浏览器，但是通过特性检测从而推断出是某个浏览器同样是很糟糕的做法。这叫做浏览器推断，是一种错误的实践。

到了某个阶段，开发者意识到 `document.all` 实际上并不是判断浏览器是否为 IE 的最佳方法。之前的代码加上了更多的特性检测，如下所示。

```
// 不好的写法  
var isIE = !!document.all && document.uniqueID;
```

这种方法属于“自作聪明”型的。尝试通过越来越多的已知特性推断某些事情太困难了。更糟糕的是，你没办法阻止其他浏览器实现相同的功能，最终导致这行代码返回不可靠的结果。

浏览器推断甚至用到了某些 JavaScript 类库中。下面的代码片段来源于 MooTools 1.1.2：

```
// 代码摘自 MooTools 1.1.2  
if (window.ActiveXObject)  
    window.ie = window[window.XMLHttpRequest ? 'ie7' : 'ie6'] = true;  
  
else if (document.childNodes && !document.all && !navigator.taintEnabled)  
    window.webkit = window>window>xpath ? 'webkit420' : 'webkit419') = true;  
else if (document.getBoxObjectFor != null || window.mozInnerScreenX != null)  
    window.gecko = true;
```

这些代码都是基于浏览器推断而确定所使用的浏览器类型，这些代码存在一些问题。

- Internet Explorer 8，同时支持 `window.ActiveXObject` 和 `window.XMLHttpRequest`，也会被识别为 Internet Explorer 7。
- 任何实现 `document.childNodes` 的浏览器，如果没有识别为 Internet Explorer 的话，都有可能识别为 WebKit 内核。
- 识别出的 WebKit 版本号太小了，再者，WebKit 422 或者更高的版本也将会被误以为是 WebKit 420^①。
- 没有检测 Opera，所以 Opera 要么被误检测为其他浏览器，要么根本不会被检测到。
- 当有新的浏览器发布时，此代码需要更新。

浏览器推断所造成的问题相当严重，尤其是最后一个。对于每一个新发行的浏览器，MooTools 都需要更新代码，并且需要相当及时地推送给用户以免代码出现问题。从长远来看这是不利于代码维护的。

要理解为什么不能使用浏览器推断，只需要回顾下高中数学课上老师讲的逻辑表达式。逻辑表达式是由假设 (p) 和结论 (q) 组成的，形如“如果 p ，那么 q ”。你可以尝试通过改变表达式的形式来确定其是否成立。这个表达式有三种变形。

- 反命题：如果 q ，那么 p 。
- 逆命题：如果非 p ，那么非 q 。
- 逆反命题：如果非 q ，那么非 p 。

各种形式的表达式之间有两个重要的关系。如果原表达式成立，那么逆反命题也成立。例如，如果原表达式为“如果它是一辆轿车，那么它有轮子”（真命题），其逆反命题“如果它没有轮子，那么它不是一辆轿车”也同样是真命题。

其次，反命题和逆命题之间也有关系，如果其中一个成立，那么另一个也一定成立。这在逻辑上是有意义的，因为反命题和逆命题间的关系同原命题和逆反命题之间的关系一样。

① 译注：原文中为 422

或许比这两种关系更重要的是并不存在的关系。如果原命题成立，不能保证反命题成立。这也正是基于特性的浏览器检测不可取的原因。

细想下这个正确的表达式：“如果浏览器是 Internet Explore，那么 `document.all` 可用。”其逆反命题，“如果浏览器没有实现 `document.all`，那么它不是 Internet Explorer”也成立。其反命题，“如果 `document.all` 可用，那么它不是 Internet Explorer。”并非严格成立（某些版本的 Opera 实现了这个特性）。基于特性的检测假定了反命题也始终成立，实际上，并没有这样的关系。

在结论中添加再多的部分也是徒劳无益的。再次看下这个表达式：“如果它是一辆汽车，那么它有轮子。”反命题很明显不成立：“如果它有轮子，那么它是一辆汽车。”你可以尝试改为更精确的表述：“如果它是一辆汽车，那么它有轮子且需要燃料。”看下它的反命题：“如果它有轮子且需要燃料，那么它是一辆汽车。”显然也不成立，因为一架飞机也符合该表述。再试一次：“如果它是一辆汽车，那么它有轮子，需要燃料，且有两个轴。”反命题也同样不成立。

现在的问题是人类语言的基础：我们无法用部分来定义整体。我们有单词“汽车”因为它隐含了很多方面的信息，否则你不得不通过穷举来确定开车上班这件事。试着通过指定越来越多的特性来确定浏览器类型也有同样的问题。你将无限接近答案，但是你永远无法得到可靠的答案。

MooTools 因为选择了基于特性的浏览器检测，从而使其自身和它的用户都陷入了困境。Mozilla 从 Firefox 3 就曾警告过 `getBoxObjectFor()` 方法将被废弃，而且会在未来的版本中去除。因为 MooTools 依靠此方法检测浏览器是否是基于 Gecko 内核的，Mozilla 在 Firefox 3.6 中将其去除，意味着任何使用旧版本 MooTools 的用户的代码至今都会受到影响。这个情况迫使 MooTools 呼吁用户升级到最新的版本，最新版本中该问题已经“Fixed”，给出的解释如下。

我们彻查了代码中的浏览器探测，改为基于用户代理字符串的检测。因为一些潜在的问题，诸如 Firefox 3.6 所示的情形，这已经在 JavaScript 类库中成为了一种标准做法。随着浏览器变得越来越接近，通过特性检测分离它们将变得越来越困难且风险很大。出于这点考虑，为了提供跨浏览器的一致体验，浏览器检测将仅在万不得已的情况下使用，这也正是人们对这一世界级的 JavaScript 框架所期待的。

12.5 应当如何取舍

特性推断和浏览器推断都是糟糕的做法，应当不惜一切代价避免使用。纯粹的特性检测是一种很好的做法，而且几乎在任何情况下，都是你想要的结果。通常，你仅需要在使用前检测特性是否可用。不要试图推断特性间的关系，否则最终得到的结果也是不可靠的。

迄今为止我不会说从来不要使用用户代理检测，因为我的确相信有合理的使用场景，但同时我也不相信会有很多使用场景。如果你想使用用户代理嗅探，记住这一点：这么做唯一安全的方式是针对旧的或者特定版本的浏览器。而绝不应当针对最新版本或者未来的浏览器。

我个人的建议是尽可能地使用特性检测。如果不能这么做的时候，可以退而求其次，考虑使用用户代理检测。永远不要使用浏览器推断，因为你会被这样维护性很差的代码缠身，而且随着新的浏览器出现，你需要不断地更新代码。

第三部分

自动化

我相当乐意花一整天的时间通过编程把一个任务实现自动化，除非这个任务手动只需要 10 秒钟就能完成。

——Douglas Adams, Last Chance to See

在 2000 年之前，Web 开发者把他们包含注释的源代码原封不动地直接部署到服务器的现象十分普遍，如果源代码有 10 个文件，在服务器上也就会有 10 个文件。这种本地和服务器文件完全一致的镜像方式可以方便我们迅速地改动代码。此外，这也导致了“查看源代码”时代的到来，许多 Web 开发者都会把查看其他网站的源代码和 Javascript 作为一种学习方式。

当然，相比现在的标准而言，那时候网站上的 Javascript 文件的数量还很少。一百行的 JavaScript 代码由一个开发人员完成几乎是常态，而如今的 Web 应用动辄就有上千行的代码，由一群或者多个开发者完成。很显然，旧的部署方式已经不能用了。

所有的大型（和许多小型的）Web 应用都依赖于自动化工具来处理它们项目中的 Javascript 文件。自动化在其他 Web 应用框架中很常见，但是直到 2005 年才开始广泛应用在 Javascript 中。把 Javascript 加入到整个 Web 应用自动化系统中可以让你拥有与系统的其他部分同样的保障，对于代码的可维护性来说，这是非常重要一步。

利弊

使用自动化构建系统构建你的 Javascript 的优点如下。

- 你本地的源代码不必同生产环境上保持一致，所以你可以任意组织你的代码结构而不必担心在服务器上使用的代码是否需要优化。
- 静态分析可以自动发现错误。
- 在部署之前有多种方式处理 Javascript，比如文件的连接和压缩。
- 通过自动化测试可以很容易地发现问题。
- 很方便地自动部署到生产环境。
- 你可以轻松快速地重新执行常见任务。

使用这样的自动化系统也会带来一些弊端。

- 开发者在开发环境每次改动后可能都需要在本地重新构建。一些习惯了改完代码就去刷新浏览器的开发人员会很难适应这一步。
- 被部署到生产的代码看起来并不像我们在开发环境编辑的代码，就使得我们很难定位线上的 Bug。
- 技术水平偏低的开发人员使用构建系统可能会遇到问题。

以我的经验看来，使用自动化的好处远远大于坏处。即便是那些讨厌改完代码还需要在本地重新编译一遍才能看到结果的人来说，一旦认识到这些好处也开始纷纷妥协了。

文件和目录结构

在开始构建你的系统之前首先要确定如何组织你的文件和目录。这个结构受项目类型影响极大。如果项目是一个独立的 JavaScript 类库，你可能需要一种与众不同的（文件和目录）结构而不是一个包含所有文件的项目网站。

13.1 最佳实践

抛开项目类型不谈，总有一些适用于 JavaScript 文件和目录结构的最佳实践。

一个文件只包含一个对象

特指每个 JavaScript 文件应该只包含一个 JavaScript 对象的代码。这种模式在其他编程语言中很常见而且通常可以使代码更易于维护。这样做降低了多个开发者同时修改同一文件的风险。尽管现在的代码版本管理工具（类似 SVN、CVS 或 GIT）在合并两个人的修改方面已经做得非常棒了，但是仍旧会有合并冲突的情况发生。文件越少，合并时候冲突的可能性就越大。保持每个文件只有一个 JavaScript 对象降低了这种风险^①。

相关的文件用目录分组

如果你有多个相关联的对象，就把这些文件都放在同一个目录下。例如，你可能有一个模块是由多个文件的代码组成的，那么把这个模块包含的所有文件放在一

^① 译注：这样做的初衷其实是让不同的人维护各自不同的文件，即完全避免多人维护一个文件的可能。

个目录下就很有意义。给相关的文件分组有助于开发者更容易定位功能（所在的代码片段）。

保持第三方代码的独立

任何不是你写的或者不是由你维护的代码都应当独立于项目工程（source control）之外。比如一个 JavaScript 类库。事实上，最理想的情况是根本就不引入这样的 JavaScript 类库，而直接从 CDN 来加载它。把这些文件单独放置在项目工程中的一个目录里，也是一个最佳的替代方案。

确定创建位置

创建后的 JavaScript 文件应该放置在一个完全独立的目录里，而且不应该提交到项目工程中。网站应当是可配置的，使用编译后的目录而非源码目录。不要把编译后的内容提交到源码工程里，这一点非常重要。因为编译后的内容都是成型的“工件”（artifacts），在最终被部署上线之前，可能会被很多人多次编译创建。部署过程应当通过编译产生可最终发布的成品，它是应当可直接部署上线的。

保持测试代码的完整性

你的 JavaScript 测试代码也应该一并提交到项目工程中的一个显眼的位置。这可以让开发人员很容易注意到遗漏测试的情况。

如果你当前工作的 JavaScript 文件是一个大型网站或者 Web 应用的一部分而不是一个独立的 JavaScript 项目，文件和目录的结构会有些许不同。整个目录结构基本上由服务端使用的框架所决定。即使整个目录结构随着项目的不同千变万化，无疑你也会有一个子目录专门用来放置 JavaScript。这个子目录可能名为 scripts 或者 javascript，但是几乎始终都有一个单独的目录专门用于存放 JavaScript 文件。针对这个目录，有几种常见的目录结构。

13.2 基本结构

JavaScript 目录下边放置如下这三个主要的目录是当下一种很流行的做法。

build

用来放置最终构建后的文件，理想情况下这个目录不应该提交。

src

用来放置所有的源文件，包括用来进行文件分组的子目录。

test 或者 tests

用来放置所有的测试文件。通常包含一些同源代码目录一一对应的子目录或文件。

我管理的 CSS Lint (<https://github.com/stubbornella/csslint>) 项目，就采用了类似这样的结构（见图 13-1）。

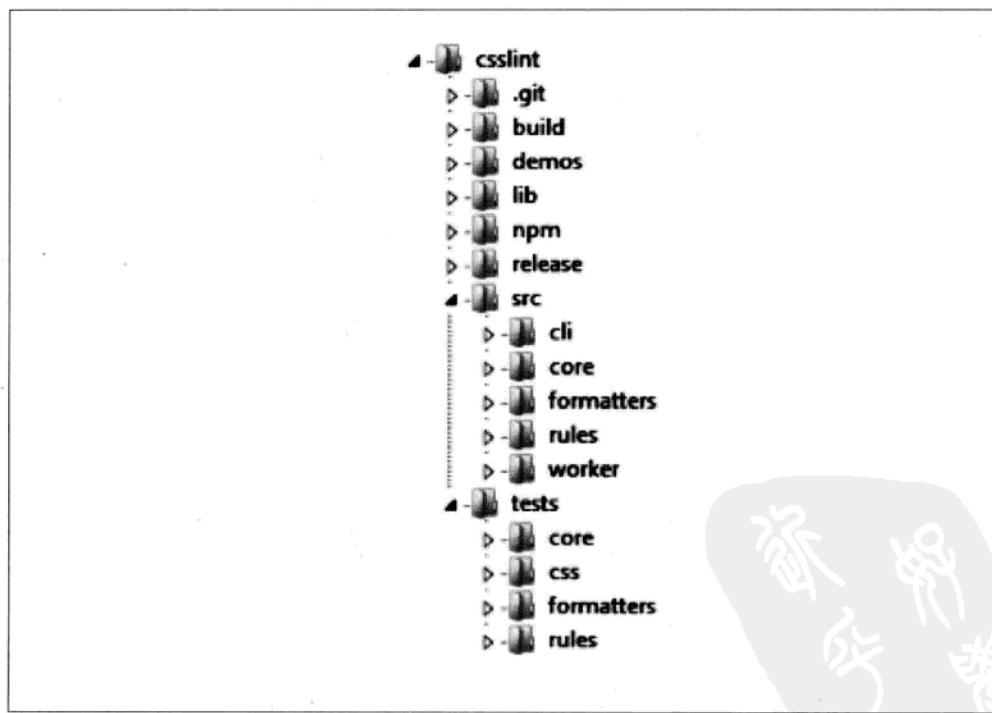


图 13-1 CSS Lint 目录结构

以 CSS Lint 为例，build 目录从不提交，release 目录始终包含最新的稳定发行版本。

src 目录下有一些按功能划分的子目录。tests 目录的结构同 src 目录的结构一一对应，所以 src/core/CSSLint.js 的测试代码即为 tests/core/CSSLint.js。

jQuery (<https://github.com/jquery/jquery>) 也采用了这种结构形式。唯一不同的就是 jQuery 把其所有的源文件都直接放在了 src 目录下，而没有每个都建立子目录。子目录保留用来存放核心特性的资源文件和扩展。test 目录存放着跟源文件名字相同的测试代码。比如 src/ajax.js 的测试代码即为 test/ajax.js (见图 13-2)。

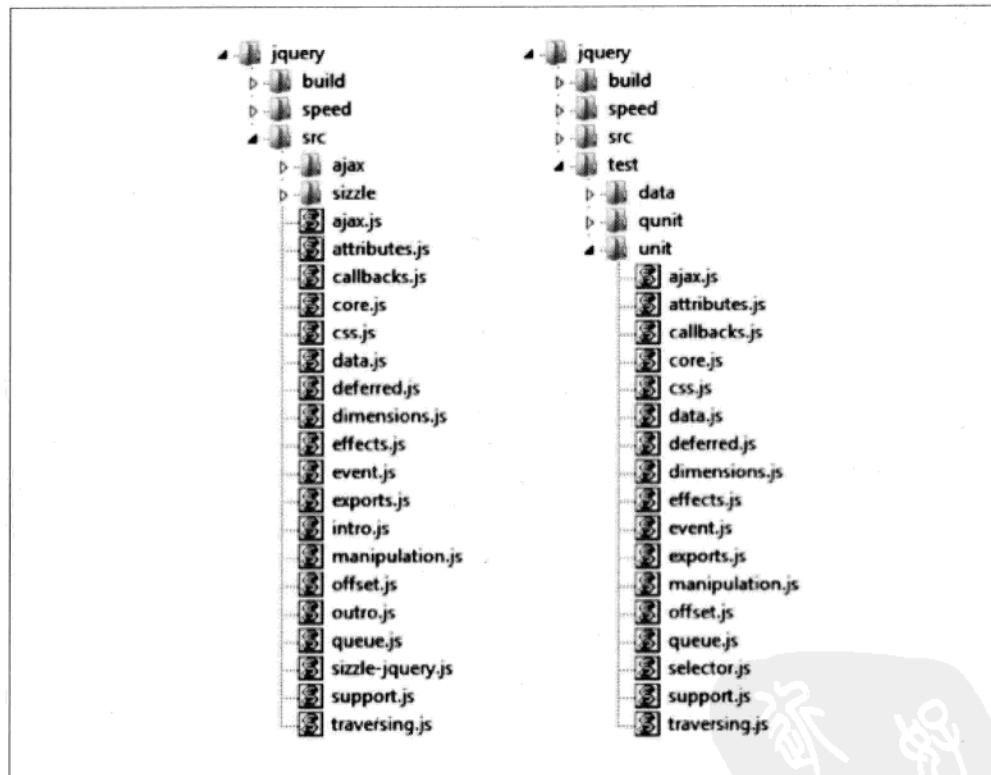


图 13-2 jQuery 目录结构

Dojo (<https://github.com/dojofoundation/dojo>) 采用了同 jQuery 类似的结构形式。其中一个很大的不同是 Dojo 根目录下没有 src 目录。而是顶层目录直接包含了所有的源文件和用来放置核心特性资源 文件和扩展的子目录。tests 目录跟顶级目录的结构一一

对应，所以 tests/data.js 即 date.js 的测试代码（见图 13-3）。

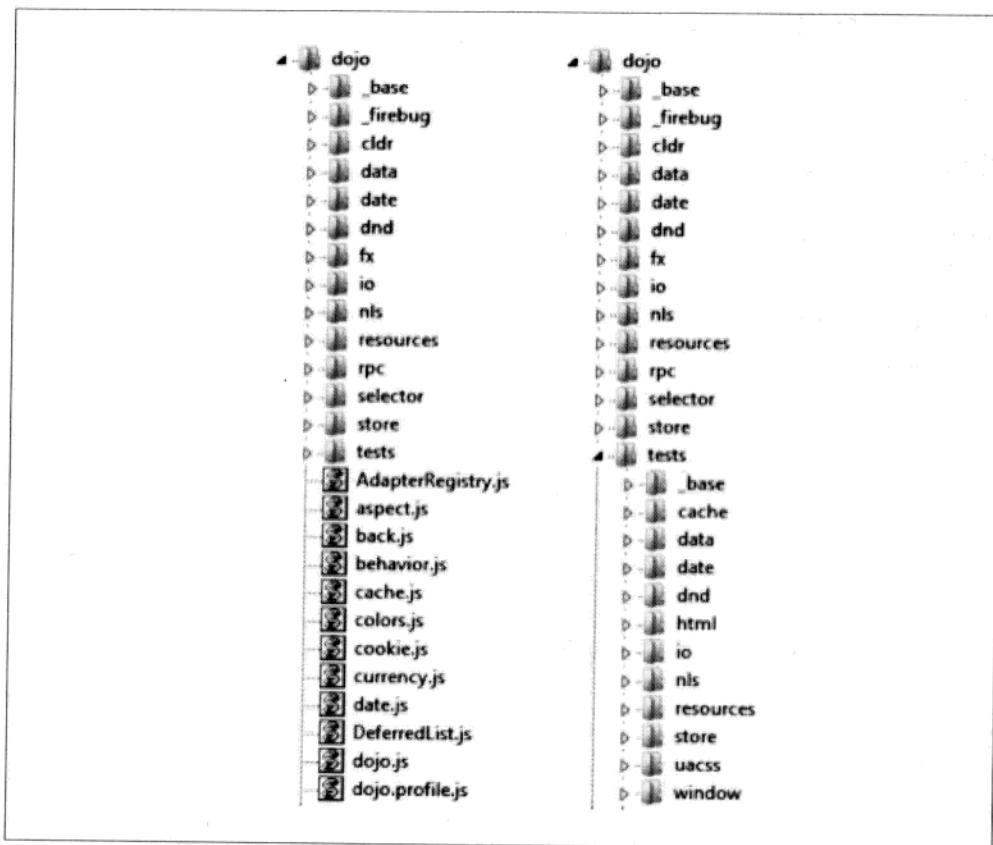


图 13-3 Dojo 目录结构

YUI3 (<https://github.com/yui/yui3>) 修改了原有的结构。src 目录下的每个子目录都代表一个单独的 YUI 模块，而且每个模块都至少有以下 4 个子目录。

docs

文档目录。

js

JavaScript 源文件目录。

meta

用以存放模块元信息（metadata）。

tests

用以存放模块的测试代码。

YUI 中的测试文件可能是 HTML 文件或者 JavaScript 文件，因此准确地说 tests 目录中包含的内容因模块而异。一般而言，至少有一个文件的名称同源文件一致。所以 tests/arraysort.html 或者 tests/array-sort.js 就是 js/arraysort.js 的测试代码（见图 13-4）。

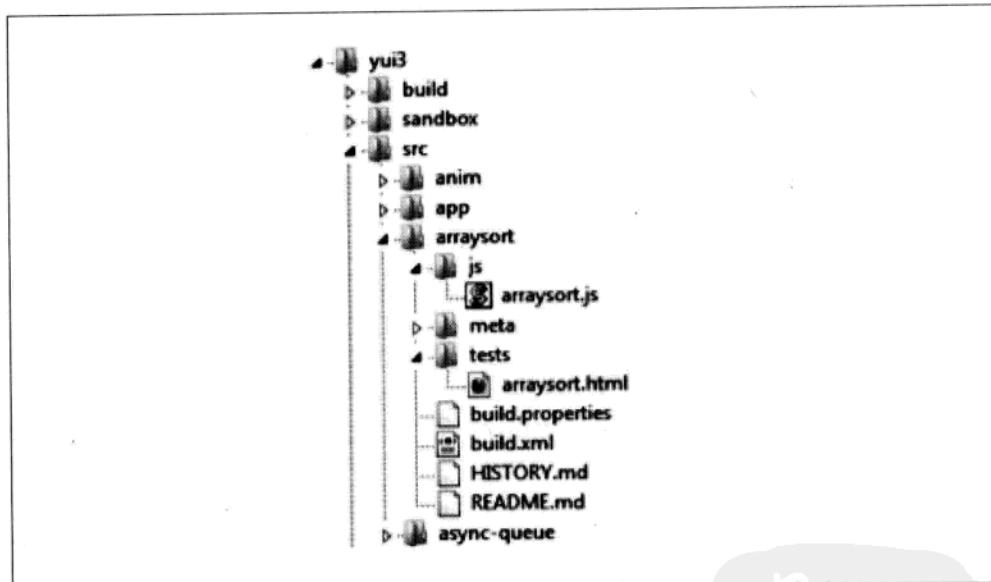


图 13-4 YUI 目录结构

准确地说你的开发环境和构建过程很大程度上决定着你会选择什么形式的结构。好的结构可以缩短构建的时间并且不会让开发人员在新建文件的时候纠结于放在哪里。

Ant

开发者通常基于他们所熟悉的工具来选择构建工具。我个人最喜欢的 JavaScript 构建工具是 Ant。Ant 最初是为 Java 项目而量身定做的构建工具，但因其简单的 XML 语法和内置的任务，它成为 JavaScript 构建工具的一个明智选择。在我写这本书之前做了一个非正式的调查，调查显示：尽管最近新的构建工具层出不穷，但是对于 JavaScript 而言，Ant 仍然是迄今为止最经常被提到的构建工具。如果你觉得 Ant 并不适合你，可以参考附录 B 中列出的一些其他选择。

这本书旨在介绍 Ant 在 JavaScript 构建系统中的使用，但其中讨论到的所有工具和技术在其他的构建系统中也同样适用。

14.1 安装

运行 Ant 需要 Java 环境，所以请确保你的系统上已经安装了 Java。如果你正在使用的是 Mac OS X 系统，则已经默认安装了 Ant；在 Ubuntu 系统中，则需要运行 `sudo apt-get install ant` 来安装 Ant。对于其他操作系统，请参照 <http://ant.apache.org/manual/install.html> 中的说明。

14.2 配置文件

Ant 主要的配置文件是 `build.xml`。当从命令行上运行 Ant 且未指定配置文件时，Ant 会在当前的工作目录中寻找这个文件，所以最好把 `build.xml` 放置在项目的根目录

下。你不必在此文件中保留所有构建相关的配置信息，但是在 Ant 运行的时候，build.xml 文件必须存在。

你可能已经猜到了，build.xml 是一个 XML 文件，用来告诉 Ant 如何执行构建过程。Ant 构建系统有三个基本组成部分。

任务

任务是构建过程中一个步骤，比如执行一个程序或者复制一个文件

目标

一组有序任务的集合

项目

所有目标的容器

构建系统的每个部分都由一个 XML 元素表示。build.xml 文件示例如下。

```
<project name="maintainablejs" default="hello">  
    <target name="hello">  
        <echo>Hello world!</echo>  
    </target>  
</project>
```

每一个 build.xml 文件都是始于一个代表整个项目的<project>元素。必须指定 name 属性，name 用来唯一标识这个项目。如果没有明确指出要执行的目标，default 属性所指示的目标就会被作为默认目标来执行。

上述这个示例文件包含一个目标，由<target>元素来表示。在这里 name 属性也是必需的。<echo>元素代表一个回显任务，它将在控制台输出所包含的文本信息。你可以在一个目标中有多个任务，也可以在一个项目中有多个目标。

一个标准的做法是尽可能颗粒化地细分目标，以便使它们可以以任何方式灵活的组合。把你想要实现功能的目标当做是重复任务按照逻辑的分组。

14.3 执行构建

只要你已经创建了 build.xml 文件，就可以在该目录下打开一个命令行，然后输入：

```
ant
```

默认情况下，Ant 会读取 build.xml 文件并读取<project>元素的 default 属性，以确定执行哪个目标。如果你用 Ant 运行了前面例子中 build.xml 文件，它会执行 hello 这一目标。你也可以在命令行上通过参数显式的指定要运行的目标。

```
ant hello
```

当你在命令行上通过参数指定目标后，Ant 将不再使用 default 属性指定的默认目标。

在这两种情况下，你都会看到控制台有这样的输出。

```
Buildfile: /home/nicholas/tmp/build.xml

hello:
[echo] Hello world!

BUILD SUCCESSFUL
Total time: 0 seconds
```

在输出结果中，第一行输出始终显示所用到的构建文件的路径，接下来是被执行的目标和被执行任务的列表。方括号里边的是任务名称，输出结果在其右边显示。接下来的一行大写显示的消息用来表示是否构建成功，最后一行是构建所用的时间。这些信息都有助于我们排查导致构建错误的原因。

14.4 目标操作的依赖

每个目标都可能会被指定依赖关系——其他被依赖的目标必须在当前目标执行之前成功执行。依赖关系通过 depends 属性来指定，多个依赖之间用半角的逗号顺序分隔，被依赖的目标将首先被执行。例如：

```
<project name="maintainablejs" default="hello">
```

```
<target name="hello">
    <echo>Hello world!</echo>
</target>

<target name="goodbye" depends="hello">
    <echo>Goodbye!</echo>
</target>

</project>
```

在这个 build.xml 文件中，目标 goodbye 依赖目标 hello。因此运行 ant goodbye 会输出下面的执行结果。

```
Buildfile: /home/nicholas/tmp/build.xml

hello:
[echo] Hello world!

goodbye:
[echo] Goodbye!

BUILD SUCCESSFUL
Total time: 0 seconds
```

由此可见，hello 目标是在 goodbye 目标之前执行的，且两者都成功地执行完毕。

在大多数的构建文件中，有一小部分的目标会被经常用到。汇总目标（rollup targets）以特定的顺序执行多个目标，大多数的目标都是设计用于汇总目标的一个步骤。

14.5 属性

Ant 中的属性类似 JavaScript 中的变量，因为它们通常都包含数据，这些数据在执行脚本期间都可以被操作和改变。比如，用<property>标签定义一个属性。

```
<project name="maintainablejs">

    <property name="version" value="0.1.0" />

</project>
```

每个<property>都需要指明 name 和 value 属性。定义后便可通过\${version}来引用，比如：

```
Version is ${version}
```

当执行这段 Ant 脚本的时候会输出：

```
Buildfile: /home/nicholas/tmp/build.xml

version:
[echo] Version is 0.1.0

BUILD SUCCESSFUL
Total time: 0 seconds
```

这个特殊的\${}语法可以让你在任何时候把你想要的属性值插入到任务中。

属性也可以定义在外部的 Java 属性文件中，并直接载入到 Ant 中。例如，假设你有一个名为 build.properties 的文件包含如下内容。

```
version = 0.1.0
copyright = Copyright 2012 Nicholas C. Zakas. All rights reserved.
```

你可以使用<loadproperties>元素并通过 srcfile 属性指明文件路径来把这些特性导入 Ant 脚本中。

```
<project name="maintainablejs" default="version">

    <loadproperties srcfile="build.properties" />

    <target name="version">
        <echo>Version is ${version}</echo>
    </target>

</project>
```

通过<loadproperties>加载的属性与那些在 build.xml 中直接定义的属性一样，都可以以相同的方式访问。对于需要定义大量的属性或是需要在多个 Ant 脚本之间公用的属性而言，最好把它们放在一个独立的 Java 属性文件中。

最好至少声明几个可用于整个项目的属性，例如：

src.dir

源代码目录的根目录。

build.dir

放置最终构建后文件的路径。

lib.dir

依赖文件的路径。

在本书接下来的部分，你会看到在很多 Ant 任务中都用到了这些属性。因此你需要在项目中合理的定义它们。

14.6 Buildr 项目

Buildr (<https://github.com/nzakas/buildr>) 是一个寻找和收集前端相关且语法简单的 Ant 任务的开源项目。虽然对于处理 JavaScript 文件而言有很多工具可以选择，但是它们之间都有一些不同。Buildr 囊括了所有这些可能在你的 Ant 脚本中使用到的各种工具，把它们封装成了任务 (tasks)。

在使用 Buildr 之前，你首先必须获得一份源代码。只要在你的电脑上有这个目录结构，你就可以用这条命令导入所有任务。

```
<import file="/path/to/buildr/buildr.xml"/>
```

通过这条命令可以在你的 build.xml 文件中使用所有 Buildr 中定义的自定义任务。

下面的章节将告诉你如何从头开始创建 Ant 任务以及如何使用 Buildr 任务。



校验

JavaScript 在被部署之前是没有经过编译的，所以 Web 开发人员不能在编译步骤来发现错误。JavaScript 代码校验器通过对代码执行静态分析在一定程度上填补了这个空白。这本书前面已经介绍了 JSLint 和 JSHint。本章主要讲解如何把 JSHint 整合到你的构建系统中来自动分析并验证你的 JavaScript 代码。

本章中之所以用到 JSHint 是因为它有适合在 Rhino 上运行的预编译命令行文件。而 JSLint，尽管一些第三方已经为 JSLint 开发了包含运行于命令行的实用程序，但是在写这本书的时候，JSLint 自身仍然没有提供预编译的命令行文件。^①

15.1 查找文件

验证文件的第一步首先就是要定位这些文件。通过两种不同的任务均可以实现这个目的：`<fileset>`和`<filelist>`。使用`<fileset>`任务可以通过通配符批量包含文件。指定 `dir` 属性为要查找的目录，然后指定 `includes` 属性为要包含的文件名表达式。例如：

```
<fileset dir=". /src" includes="**/*.js" />
```

上述 `fileset` 包含了 `src` 目录中所有的 JavaScript 文件，你也可以通过指定文件名表达式选择性地排除一些文件。

^① 译注：除了在预编译阶段来控制语法检查和校验，在源码编辑的时候也可以实时监控源码的错误，同样，JSLint 也有基于 JavaScript 的版本，也是可以运行于 Rhino 上的，这里有一个例子，基于 vim 的 js 语法检查插件：<http://ued.taobao.com/blog/2010/11/11/jshint-for-vim/>。

```
<fileset dir=".src" includes="**/*.js" excludes="**/-test.js" />
```

上述 `fileset` 包含除了文件名以 `-test.js` 结尾的文件以外的所有 JavaScript 文件。这是一种很常见的做法，用于排除源代码目录中包含的单元测试文件。

`<filelist>` 任务的工作原理是相似的，不同的是它必须要明确指定要包含的文件。当你想要引用一组特定的文件时，这项任务是最好用的。`<filelist>`^① 元素同样需要 `dir` 属性来指定目录。`files` 属性包含一组以逗号分隔的文件列表。例如：

```
<filelist dir=".src" files="core/core.js" />
```

在实际情况中，你最终将会使用 `<fileset>` 比 `<filelist>` 更频繁，因为你更可能要处理大批文件而不是几个特定的文件。

15.2 任务

`<apply>` 任务用于对一个 Ant 目标里的文件集合执行命令行程序。因为 JSHint 是用 JavaScript 语言编写，所以你需要使用 Rhino 命令行 JavaScript 引擎来执行它。从 <http://www.mozilla.org/rhino> 下载最新发布的 Rhino，并把 `js.jar` 文件放在你的依赖文件夹里 (`lib.dir`)。

若要在命令行上运行 JSHint，键入：

```
java -jar js.jar jshint.js [options] [list of files]
```

例如：

```
java -jar js.jar jshint.js curly=true,noempty=true core/core.js
```

`<apply>` 任务可以让你使用 `<arg>` 元素来重建命令行条目。使用 `<arg>` 有两种方式：通过 `path` 属性指定引用的文件或者目录，或者指定纯文本的 `line` 属性。可以把命令行格式分解成以下几部分。

```
java
```

要执行的程序，由 `<apply>` 的 `executable` 属性来指定。

① 译注：原文中为 `<fileset>`，应该是作者笔误

```
-jar
```

用于 java 的选项，由<arg>的 line 属性来指定。

```
jshint.js
```

JSHint 的主要文件，由<arg>的 path 属性来指定。

```
curly=true,noempty=true
```

附加选项，由<arg>的 line 属性来指定。

```
core/core.js
```

要验证的文件，由<arg>的 path 属性来指定。

基于此，你就可以快速为这个实用命令行程序创建一个 Ant 目标的梗概。

```
<target name="validate">
  <apply executable="java">
    <arg line="-jar"/>
    <arg path="js.jar"/>
    <arg path="jshint.js" />
    <arg
      line="curly=true,forin=true,latedef=true,noempty=true,undef=true,
rhino=false"
    />
    <arg path="core/core.js"/>
  </apply>
</target>
```

尽管这种方法很有效，但是你应该真正需要用 JSHint 处理的是一个 JavaScript 文件集而不是单个的 JavaScript 文件。<apply>任务很容易实现这一点，你可以指定一个<fileset>并且使用<srcfile>元素把它放入命令行的特定位置。例如，以下目标就是为了对源目录中所有的 JavaScript 文件执行校验。

```
<target name="validate">
  <apply executable="java">
    <fileset dir="${src.dir}" includes="**/*.js" />
    <arg line="-jar"/>
    <arg path="js.jar"/>
    <arg path="jshint.js" />
    <arg
```

```
    line="curly=true,forin=true,latedef=true,noempty=true,undef=true,
rhino=false"
  />
  <srcfile/>
</apply>
</target>
```

现在这个 Ant 目标中通过<fileset>元素指定的每一个文件上都被执行了 JSHint 校验。你可以通过下述代码来运行目标。

```
ant validate
```

15.3 增强的目标操作

尽管上述 validate 目标工作得很好，但是其实是可以做一些改进的。首先，现有版本是对每个文件都执行一次 JSHint 的。所以倘若有 3 个 Javascript 文件，就等同于执行了 3 条命令。

```
java -jar js.jar jshint.js curly=true,noempty=true first.js
java -jar js.jar jshint.js curly=true,noempty=true second.js
java -jar js.jar jshint.js curly=true,noempty=true third.js
```

当运行 java 时是有一些额外开销的，尤其是在 Java 虚拟机（Java Virtual Machine, JVM）创建和结束的时候。所以这个任务在执行目标的时候会耗费大量的时间。

实际上 JSHint 命令行脚本是支持接收多个文件作为参数的，因此理想情况下是使用一个 JVM 来检查每个文件。好在<apply>任务可以让这些文件名的传递更加简单。你仅仅需要设置 parallel 属性为 “true”。

```
<target name="validate">
  <apply executable="java" parallel="true">
    <fileset dir="${src.dir}" includes="**/*.js" />
    <arg line="-jar"/>
    <arg path="js.jar"/>
    <arg path="jshint.js" />
    <arg line="curly=true,forin=true,latedef=true,noempty=true,undef=true,
rhino=false" />
    <srcfile/>
  </apply>
</target>
```

通过添加一个属性，`validate` 目标现在会在命令行把所有文件一次性传递（之间通过空格隔开）。这么做可以使 JSHint 通过一个 JVM 读取、校验所有文件，从而可以显著地提升目标的执行速度。

最后一个针对 `validate` 目标的改动是当验证失败时候使构建失败。通常添加这一步是一个好主意，因为这样做能确保开发者意识到问题所在。当前版本的 `validate` 目标将会在命令行输出验证失败的信息，但是其后的任务仍然会继续执行，这可能导致失败信息滚动到屏幕之外。

你可以通过将 `failonerror` 属性设置为“`true`”，强制`<apply>`出错时构建失败。

```
<target name="validate">
    <apply executable="java" failonerror="true" parallel="true">
        <fileset dir="${src.dir}" includes="**/*.js" />
        <arg line="-jar"/>
        <arg path="js.jar"/>
        <arg path="jshint.js" />
        <arg line="curly=true,forin=true,latedef=true,noempty=true,undef=true,
rhino=false" />
        <srcfile/>
    </apply>
</target>
```

现在这个版本的 `validate` 目标在执行`<apply>`出错时将导致构建失败。执行出错时将会返回非零的退出码。因为 JSHint 在校验出错时会返回 1，所以将导致构建失败。

15.4 其他方面的改进

改进 `validate` 目标的最后一步就是将下面这三部分数据抽离出来，放于外部。

- `js.jar` 的位置。
- `jshint.js` 的位置。
- 命令行选项。

由于这些值在未来可能会改变，所以最好通过属性的方式定义它们，并且在你的属

性文件或者 build.xml 文件顶部引入。下面是一个属性文件的例子。

```
src.dir = ./src
lib.dir = ./lib

rhino = ${lib.dir}/js.jar
jshint = ${lib.dir}/jshint.js

jshint.options = curly=true,forin=true,latedef=true,noempty=true,undef=true\
,rhino=false
```

接着你就可以在 validate 目标中引用这些属性值。

```
<target name="validate">
<apply executable="java" failonerror="true" parallel="true">
<fileset dir="${src.dir}" includes="**/*.js" />
<arg line="-jar"/>
<arg path="${rhino}"/>
<arg path="${jshint}"/>
<arg line="${jshint.options}" />
<srcfile/>
</apply>
</target>
```

通过这种方式，你不用再返回到目标就能够轻松地更新文件的位置和 JSHint 选项。

15.5 Buildr 任务

Buildr 有一个<jshint>任务，把很多运行 JSHint 所必需的配置抽离了出来。在导入前一章中提到的 buildr.xml 文件之后，通过一些<fileset>元素即可使用<jshint>任务。

```
<target name="validate">
<jshint>
<fileset dir="${src.dir}" includes="**/*.js" />
</jshint>
</target>
```

也可以使用 options 属性来更改默认选项。

```
<target name="validate">
  <jshint options="${jshint.options}">
    <fileset dir="${src.dir}" includes="**/*.js" />
  </jshint>
</target>
```

最终的结果与上一节中使用的目标完全一致。

第 16 章

文件合并和加工

如果你恰当地组织你的 JavaScript 文件——每个文件包含一个对象，那么你可能有许多个 JavaScript 文件。在部署到生产环境之前，最好把这些文件合并在一起，这样可以减少页面上的 HTTP 请求数。在一个项目中，究竟有多少文件，哪些文件需要合并，这要视具体情况而定。无论如何，Ant 提供了一个简单的方法来合并多个文件。

16.1 任务

在 Ant 中，`<concat>`任务是最简单的任务之一。你只需指定包含目标文件名的 `destFile` 属性，然后包含你想要用到的`<fileset>`和`<filelist>`元素。简而言之，你可以像下面这样设置一个目标。

```
<target name="concatenate">

    <concat destfile="${build.dir}/build.js">
        <fileset dir="${src.dir}" includes="**/*.js" />
    </concat>

</target>
```

这个目标将把所有源目录中的 JavaScript 文件合并在一起，并在构建目录下生成单个 `build.js` 文件。需要注意的是这些文件是按照它们在文件系统中的顺序（按照字母排序）有序连接的。如果你需要一个更特殊的顺序，就需要显式地指定它。例如：

```
<target name="concatenate">

    <concat destfile="${build.dir}/build.js">
        <filelist dir="${src.dir}" files="first.js,second.js" />
        <fileset dir="${src.dir}" includes="**/*.js" excludes="first.js,
second.js"/>
    </concat>

</target>
```

这个版本的目标操作是将 `first.js` 作为第一个文件合并入最终文件，`second.js` 紧随其后。记住，你可以使用任意数目的`<fileset>`和`<filelist>`元素，因此你能够以任意顺序合并这些文件。

尽管使用 Ant 也许可以创建复杂的合并方案，但是最好限制使用这些特殊情况。对于提升可维护性来说，不要在构建脚本内保持和文件名的耦合，这是一种最佳实践。尽可能试着使用`<fileset>`元素。

16.2 行尾结束符

把这些文件合并在一起，会遇到一系列挑战。其中最棘手的问题就是处理文件的最后一行。如果一个文件最后一行没有换行符，那么把这个文件与其他文件合并的话，可能会导致语法错误。设置 `fixlastline` 属性为 “yes”，`<concat>`任务就会在最后一行没有换行符的时候自动添加一个换行符。

```
<target name="concatenate">

    <concat destfile="${build.dir}/build.js" fixlastline="yes">
        <filelist dir="${src.dir}" files="first.js,second.js" />
        <fileset dir="${src.dir}" includes="**/*.js" excludes="first.js,
second.js"/>
    </concat>

</target>
```

在 JavaScript 中，默认设置 `fixlastline` 为 “yes” 是很好的做法，因为换行符是有效的语句结束指令。

修复“最后一行”是很有用的，但我怎么才能知道使用了什么行尾标识符呢？

如果不同的人在不同的操作系统上编辑你的源文件，那么你肯定希望确保构建文件跨平台的一致性。`<concat>`任务有一个可选的 `eol` 属性，这一属性指定了要使用哪个行尾标识符。默认值是在操作系统中的默认行尾标记（在 Windows 中默认为“crlf”，在 UNIX 中默认为“lf”，在 Mac OS X 中默认为“cr”）。在这些属性值中，你可以任意选择一个来设置属性，并且连接后的文件中的所有行尾会自动切换到该格式^①。

```
<target name="concatenate">

    <concat destfile="${build.dir}/build.js" fixlastline="yes" eol="lf">
        <filelist dir="${src.dir}" files="first.js,second.js" />
        <fileset dir="${src.dir}" includes="**/*.js" excludes="first.js,
second.js"/>
    </concat>

</target>
```

这个目标把所有行尾标记改为 UNIX 格式，这是 JavaScript 文件的推荐格式，因为它有良好的跨平台兼容性（并且大多数 Web 服务都运行在 UNIX 环境上）。

16.3 文件头和文件尾

`<concat>`任务还能很方便地把纯文本“前置”（`prepend`）和“追加”（`append`）到生成的文件中。有了这个功能，你就可以把那些可能还没有用到的信息插入到文件中。例如，我倾向于把构建时间插入到文件中，这样我就可以更容易追踪错误。因此，我需要先在文件顶部定义一个`<tstamp>`元素。

```
<tstamp>
    <format property="build.time"
        pattern="MMMM d, yyyy hh:mm:ss"
        locale="en,US"/>
</tstamp>
```

当 Ant 执行 `build.xml` 文件时，这段代码就会创建一个新的时间戳。由此产生的日期字符串存储在名为 `build.time` 属性中。`pattern` 属性是一个日期时间格式字符串。

① 译注：在 Windows 操作系统里文本文件的行尾为回车加换行（0A0D），在 UNIX 中文本行尾只需要一个换行（0D）即可，当 Windows 下的文本文件在 UNIX 中打开时，行尾的 0A（回车）就被显示出来了，显示为控制字符，比如在 vim 中就会看^M 之类的标识符，如下文所说，推荐使用 UNIX 的文件格式。

然后我可以用`<header>`元素把这个信息添加到构建文件中。

```
<target name="concatenate">

    <concat destfile="${build.dir}/build.js" fixlastline="yes" eol="lf">
        <header>/* Build Time: ${build.time} */</header>
        <filelist dir="${src.dir}" files="first.js,second.js" />
        <fileset dir="${src.dir}" includes="**/*.js" excludes="first.js,
second.js"/>
    </concat>

</target>
```

这个新版本的 `concatenate` 目标在文件顶部插入了一个包含构建时间的注释。因此，最终文件的第一行就是这样的格式。

```
/* Build Time: May 25, 2012 03:20:45 */
```

另外，还有一个`<footer>`元素，可以把额外的文本添加到文件底部。例如，这个版本的 `concatenate` 目标把构建时间放在了底部。

```
<target name="concatenate">

    <concat destfile="${build.dir}/build.js" fixlastline="yes" eol="lf">
        <filelist dir="${src.dir}" files="first.js,second.js" />
        <fileset dir="${src.dir}" includes="**/*.js" excludes="first.js,
second.js"/>
        <footer>/* Build Time: ${build.time} */</footer>
    </concat>

</target>
```

你可以同时使用`<header>`和`<footer>`，当然，也可以一次只使用其中一个。

16.4 加工文件

加工是指在确认并准备好可以部署之前对文件的最后修改。很多时候，这一步骤涉及把额外的文本添加一个文件中，又或者是用别的一些东西替代现有的文本。如同前面的例子中，插入构建时间就是加工的其中一种形式。其他常见的任务比如自动引入许可信息和插入版本信息。使用 Ant，两者都可以很容易地实现。

许多项目在源代码目录的某处都有一个 license 文件。license 文件是单独的，因为它可能会改变代码的独立性。在文件被推送到生产环境之前，自动把许可信息插入到文件顶部是很有用的。你也许可以使用<filelist>元素插入许可文件，但是这意味着 license 文件内容必须以 JavaScript 注释的格式存放。不过在纯文本格式的 license 文件周围添加注释是很容易的。你可以使用<loadfile>任务从任何文件中加载文本。

```
<loadfile property="license" srcfile="license.txt" />
```

这行代码从 license.txt 中加载文本，并且将它存储在名为 license 的属性中。一旦它在一个属性中，你就可以使用<header>元素来把文本作为注释插入。

```
<target name="concatenate">

    <loadfile property="license" srcfile="license.txt" />

    <concat destfile="${build.dir}/build.js" fixlastline="yes" eol="lf">
        <header trimleading="yes">/*
            ${license}
        */
        /* Build time: ${build.time} */
        </header>
        <filelist dir="${src.dir}" files="first.js,second.js" />
        <fileset dir="${src.dir}" includes="**/*.js" excludes="first.js,
second.js"/>
    </concat>

</target>
```

这样一来，license 文件就作为 JavaScript 多行注释插入进来了，以一个感叹号作为第一个字符，是用来告诉压缩工具（见第 17 章）这个注释很重要且不应当删除。<header>元素中把 trimleading 设置为 “yes”。这个属性指定了在<header>里的每一行的最前边是不可以有空格的。这样一来，最后文件中所有的文本都要排第一列。

加工的其他部分——在文件中替换一些文本，使用<replaceregexp>任务就可以很容易完成。此任务会系统地处理任意数量的文件，并且使用正则表达式来进行值的替换。例如，在我的源文件中，我更喜欢用@VERSION@标记来表示版本号应该插入到什么位置。举例说明，在一个 JavaScript 文件中，可能会有像下面

这样的情况。

```
var MyProject = {  
    version: "@VERSION@"  
};
```

你可以用下面的方法来替换@VERSION@为一个确切的版本号。

```
<replaceregexp match="@VERSION@" replace="${version}" flags="g" byline="true">  
    <fileset dir="${build.dir}" includes="**/**"/>  
</replaceregexp>
```

<replaceregexp>任务将从 match 属性中取出正则表达式，并且用 replace 属性中指定的文本来替换它。正则表达式法的标志例如 g、i 和 m 是由 flags 属性指定的，byline 属性表明了这个正则表达式是否应该只匹配一行。然后，你可以指定任意数量需要被替换的文件。

由于<replaceregexp>没有创建新的文件，因此要确保是在构建文件上执行，如下面的例子所示。

```
<target name="concatenate">  
  
    <concat destfile="${build.dir}/build.js" fixlastline="yes" eol="lf">  
        <filelist dir="${src.dir}" files="first.js,second.js" />  
        <fileset dir="${src.dir}" includes="**/*.js" excludes="first.js,  
second.js"/>  
        <footer>/* Build Time: ${build.time} */</footer>  
    </concat>  
  
    <replaceregexp match="@VERSION@" replace="${version}" flags="g"  
    byline="true">  
        <fileset dir="${build.dir}" includes="**/**"/>  
    </replaceregexp>  
  
</target>
```

以上代码用 version 属性的值替换了所有构建文件中的@VERSION@。尽管在这里替换操作是包含在 concatenate 目标中的，但是你可能希望一个单独的目标用来专门做加工，例如：

```
<target name="bake">  
  
    <replaceregexp match="@VERSION@" replace="${version}" flags="g"
```

```
byline="true">
    <fileset dir="${build.dir}" includes="**/*"/>
</replaceregexp>

</target>
```

当加工过程没有涉及`<concat>`任务，连接的操作可能也不作为构建过程的一部分，这时，将加工步骤进行分解就显得很有意义了。



文件精简和压缩

当你完成了文件的构建、验证、合并和加工，就到了尽可能把文件变小的时候了。完成这一步需要两步：文件精简和压缩。文件精简是指消除不必要的空格，去除注释，并对文件做一些处理使其体积尽可能的小。压缩是指使用特定的压缩方法进一步缩小文件的体积，例如 gzip。文件精简和压缩之间的区别在于，处理后的文件是否仍然只是纯文本，并且可以像通常一样编辑或加载（虽然因为所有格式都已经被去除而看上去有些难读），而压缩后的文件是不可读的并且在网页中必须先要解压后才能使用。如今的浏览器在响应头信息中接收到 Content-Encoding: gzip 的时候会自动解压这些接收到的文件。

17.1 文件精简

精简（minification）一个 JavaScript 文件并不复杂^①，但是如果使用不当可能会导致错误或者无效的语法。出于这个原因，实际使用中最好在使用压缩工具压缩文件之前对 JavaScript 语法进行解析。解析器能够有效地识别出 JavaScript 语法并且可以更容易地创建有效的语法。三种最受欢迎的解析型压缩工具分别如下。

^① 译注：文件的压缩精简本身并不复杂，但对于带有中文的 JavaScript 文件来说，仅做源码压缩是不够的，还需要对文件做 Unicode 转码，以保证压缩后的文件不会因为编码问题引入 bug，此外，由于每个项目 JS 文件较多，也需要一些批量压缩工具来提高效率，这里推荐译者开发的一个工具 TPacker：<http://github.com/jayli/TPacker>。

YUI Compressor

YUI Compressor 被认为是最受欢迎的基于解析器的而不是基于正则表达式的压缩工具。 **YUI Compressor** 最初由 Julien Lecomte 编写（现在由 YUI 团队维护），它将去除代码中的注释和额外的空格并且会用单个或两个字符来代替局部变量用以节省更多的字节。**YUI Compressor** 认为语法和运行期的安全性是优先级最高的，因此默认情况下是关闭了可能导致错误的变量替换（比如使用 eval() 或者 with）。从可以下载 **YUI Compressor**。

Closure Complier

Closure Complier 也是一个基于解析器的压缩工具，它会试图让你的代码变得尽可能的小。**Closure Complier** 是由 Google 的工程师编写并维护的，它会去除注释和额外的空格并进行变量替换，而且还会通过分析你的代码来进行相应的优化。例如：它可以检测并删除定义但未被使用的函数。它还会将检测到的只用到过一次的函数转变为内联函数。出于这个原因，最好使用 **Closure Complier** 一次性检查你所有的 JavaScript 代码。下载链接：<http://code.google.com/closure/complier/>。

UglifyJS

UglifyJS 被认为是第一个基于 Node.js 的压缩工具，**UglifyJS** 是由 Mihai Bazon 用 JavaScript 编写的基于 JavaScript 的解析器。**UglifyJS** 会去除注释和额外的空格，替换变量名，合并 var 表达式，并进行一些其他方式的优化。可以从 <https://github.com/mishoo/UglifyJS> 下载，也可以使用 npm 安装。

究竟选择哪个压缩工具完全见仁见智。一些人喜欢 **YUI Compressor**，因为它确保压缩后的代码没有错误并且会做简单优化。另外一些人喜欢 **Closure Complier**，因为它处理后的文件比 **YUI Compressor** 处理后的更小。然而还有一些人喜欢 **UglifyJS**，因为它不依赖 Java 环境，相比 **Closure Complier** 而言产生的语法错误更少，而且处理后的文件可能更小。^①

① 译注：除了这三个压缩工具之外，还有一个较为常用的压缩工具 **JSmin**，它是所有 js 压缩工具的鼻祖，作者是 Douglas Crockford，**JSmin** 只会简单地去除空格和回车，不会对 js 文件进行语法上的修改，因此也是最安全的压缩工具，但 **JSmin** 的压缩比不如文中提到的这三个压缩工具。

17.1.1 使用 YUI Compressor 精简代码

YUI Compressor 作为一个可执行的 JAR 包文件必须放置在项目的依赖目录下（在这个例子中，这个目录指的是 lib.dir）。其中有许多的命令行参数，但是要了解的最重要的几个如下。

```
--disable-optimizations
```

关闭细微优化，例如把 obj["prop"] 转化为 obj.prop。

```
--line-break <column>
```

在指定的列处换行，而不是生成的输出结果只有一行。

```
--nomunge
```

关闭局部变量名称替换。

```
--preserve-semi
```

关闭对不必要的分号的去除。

一旦当你决定使用哪些参数，就可以在 Java 属性文件中配置，比如下面的例子。

```
src.lib = ./src
lib.dir = ./lib

yuicompressor = ${lib.dir}/yuicompressor.jar

yuicompressor.options = --preserve-semi
```

接下来，创建一个文件精简的目标。像验证一样，首先看下命令行的语法是大有裨益的。键入如下命令在命令行运行 YUI Compressor。

```
java -jar yuicompressor.jar [options] [file] -o [outputfile]
```

例如：

```
java -jar yuicompressor.jar --preserve-semi core/core.js -o core/core-min.js
```

该命令将通过 YUI Compressor 对 core/core.js 执行压缩并输出结果到 core/core-min.js.

精简文件时给文件名添加-min 后缀是一个最佳实践。这个目标又一次用到了<apply>任务。

```
<target name="minify">

    <apply executable="java" failonerror="true">
        <fileset dir="${build.dir}" includes="*.js"/>
        <mapper type="glob" from="*.js" to="${build.dir}/*-min.js"/>

        <arg line="-jar"/>
        <arg path="${yuicompressor}"/>
        <arg line="${yuicompressor.options}"/>
        <srcfile/>

        <arg line="-o"/>
        <targetfile/>
    </apply>

</target>
```

这个目标跟书中先前提到的 validate 目标很是类似。开始指定了可执行文件为 java，而后指明了该命令应当对构建目录下的所有 JavaScript 文件均执行。这些文件一个接一个地准备就绪并替换在<srcfile/>位置。下一行使用了<mapper>任务来转换文件名。它将给任意的<srcfile/>为文件名添加-min 后缀。因此 core.js 将变为 core-min.js，并且该值将取代<targetfile/>元素。其余<arg>元素都是不言而喻的，在此不再赘述。

minify 任务原本是用来在你生成构建文件到构建目录之后使用的，但是稍作修改就可以在任意目录执行。

YUI Compressor 也包含一个 CSS 精简工具。如果你给 YUI Compressor 传递的是一个 CSS 文件，它会默认切换为 CSS 模式。

Buildr 有一个<yuicompressor>任务封装了所有的功能。每个命令行参数都有对应的属性（属性名称跟命令行参数名相同，不含--），另外有一个必须的 outputdir 属性指明精简后文件的放置目录。举个例子：

```
<target name="minify">
    <yuicompressor outputdir="${build.dir}" preserve-semi="true">
        <fileset dir="${build.dir}" includes="*.js" />
```

```
</yuicompressor>  
</target>
```

<yuicompressor>任务自动会给所有创建的文件添加-min 后缀。你可以通过一个或多个元素一次性添加所有需要精简的文件。

17.1.2 用 Closure Compiler 精简

Closure Compiler 也是一个可执行的 JAR 文件并需要放置在依赖目录下。Closure Complier 相比 YUI Compressor 而言增加了一些值得关注的选项，其中许多仅用于直接使用 Closure JavaScript 库的代码。最重要的选项莫过于--compilation_level，这决定了对 JavaScript 文件处理的程度。有如下选项。

WHITESPACE_ONLY

仅仅删除不必要的空格和注释。关闭其他优化。

SIMPLE_OPTIMIZATIONS

Closure Complier 的默认设置。该设置将移除不必要的空格和注释但是依旧会将局部变量重命名为短名称。即使在 eval() 和 with 语句中，也会进行重命名，因此如果使用了任意一个，它都可能会导致运行期的错误。

ADVANCED_OPTIMIZATIONS

每一个优化都可能在代码中完成。谨慎使用，因为这可能会引起运行期错误或者语法错误。

当你决定了要使用的选项，可以将它们写在 Java 属性文件中，例如：

```
src.dir = ./src  
lib.dir = ./lib  
  
closure = ${lib.dir}/compiler.jar  
  
closure.options = --compilation_level SIMPLE_OPTIMIZATIONS
```

若要在命令行执行 Closure Complier，键入如下命令。

```
java -jar compiler.jar [options] --js [file] --js_output_file [outputfile]
```

例如：

```
java -jar compiler.jar --compilation_level SIMPLE_OPTIMIZATIONS --js
core/core.js
--js_output_file core/core-min.js
```

该命令将对 core/core.js 执行 Closure Complier 并将结果输出到 core/core-min.js。其 Ant 目标同使用 YUI Compressor 的目标基本相同。

```
<target name="minify">

<apply executable="java" failonerror="true">

    <fileset dir="${build.dir}" includes="*.js"/>
    <mapper type="glob" from="*.js" to="${build.dir}/*-min.js"/>

    <arg line="-jar"/>
    <arg path="${closure}"/>
    <arg line="${closure.options}"/>

    <arg line="--js"/>
    <srcfile/>

    <arg line="--js_output_file"/>
    <targetfile/>
</apply>

</target>
```

除了改变了 JAR 文件的路径和命令行的参数之外，这个目标同 minify 目标几乎相同，最终结果也一样：每个在构建目录下的 JavaScript 文件都被精简并且输出一个含有-min 后缀的新文件。

Buildr 有一个<closure>任务封装了所有这些功能。你可以使用 compilation-level 属性设置编译等级。同<yuicompressor>类似，outputdir 属性是必须的，用来指明精简后的文件放置的目录。举个例子：

```
<target name="minify">
    <closure outputdir="${build.dir}" compilation-level="SIMPLE_
OPTIMIZATIONS">
        <fileset dir="${build.dir}" includes="*.js" />
    </closure>
</target>
```

<closure>任务自动为所有生成的文件添加-min 后缀。你可以包含一个或多个元素

一次性精简所有文件。

17.1.3 使用 UglifyJS 精简

通过 npm (Node.js 包管理), UglifyJS 是在命令行最常用到的。在你安装 UglifyJS 之前, 你必须首先安装有 Node.js 和 npm, 下述命令可以完成安装。

```
sudo npm install -g uglify-js
```

UglifyJS 的命令行接口也有很多选项。下边是最常用到的一些。

```
--beautify
```

对 js 代码进行重新格式化而不是压缩它。

```
--no-mangle
```

关闭函数名和变量名的替换。

```
--no-mangle-functions
```

仅仅关闭函数名替换。

```
--no-dead-code
```

移除不可读的代码。

UglifyJS 在命令行的基本格式如下。

```
uglifyjs [options] -o [outputfile] [file]
```

例如:

```
uglifyjs --no-mangle-functions -o core/core-min.js core/core.js
```

这条命令将对 core/core.js 执行 UglifyJS 并且输出结果到 core/core-min.js。很重要的
是原文件在所有其他选项之后。

如同其他压缩工具一样, 最好将你用到的选项写进属性文件中。例如:

```
src.dir = ./src
lib.dir = ./lib

uglifyjs = uglifyjs

uglifyjs.options = --no-mangle-functions
```

UglifyJS 的 Ant 目标较其他而言略微简单些，因为它是一个独立的可执行文件。与先前给<apply>设置 executable 属性为 Java 不同的是，设置 executable 为 uglifyjs。附加信息同先前一样。

```
<target name="minify">
    <apply executable="uglifyjs" failonerror="true">

        <fileset dir="${build.dir}" includes="*.js"/>
        <mapper type="glob" from="*.js" to="${build.dir}/*-min.js"/>

        <arg line="${ugilfyjs.options}"/>
        <arg line="-o"/>
        <targetfile/>
        <srcfile/>
    </apply>

</target>
```

minify 目标的格式同本章中的其他目标一样，并且有类似的输出结果。构建目录下的所有文件都会被压缩并且生成-min 后缀的文件。

Buildr 的<uglifyjs>任务可以让 UglifyJS 的使用更简单。每一个命令行参数都有对应的属性（不含开头的--），跟其他压缩任务一样。outputdir 属性是必不可少的。举例说明。

```
<target name="minify">
    <uglifyjs outputdir="${build.dir}" no-mangle-functions="true">
        <fileset dir="${build.dir}" includes="*.js" />
    </uglifyjs>
</target>
```

<uglifyjs>任务同其他任务工作方式相同：自动为压缩后的文件名添加-min 后缀，如果你需要，也可以通过指定多个<fileset>元素。

17.2 压缩

JavaScript 文件的精简（minification）是部署之前的第一步。接下来要尽可能把文

件压缩（compression）到最小以利于文件传输。本章中提到的压缩工具并未对 JavaScript 执行压缩（YUI Compressor 甚至仅仅执行文件精简操作）。压缩通常在这些处理之后，在运行期通过 Web 服务器使用 HTTP 压缩或者在构建期执行压缩。

17.2.1 运行时压缩

大多数 Web 服务器都支持运行时执行文件压缩。实际情况中，这些压缩一般仅仅适用于基于文本的文件，像 JavaScript、HTML 和 CSS。现代浏览器都支持 HTTP 压缩并且都会发送一个 HTTP 头作为请求的一部分，指明压缩类型。例如：

```
Accept-Encoding: gzip, deflate
```

当服务端在 HTTP 请求中发现上述头信息，它就会明白浏览器是支持通过 `gzip` 或者 `deflate` 对压缩后的文件执行解压缩的。当服务器发送响应时，它会设置头信息指明压缩的类型，例如：

```
Content-Encoding: gzip
```

该头信息告知浏览器响应正文是通过 `gzip` 压缩过的并且必须解压缩才能使用。

Apache 2，最受欢迎的 Web 服务器之一，内置了 HTTP 压缩作为 `mod_deflate` 模块。该模块默认开启且自动压缩 JavaScript、HTML、CSS 和 XML 文件。如果你在使用 Apache 2，你不需要针对 JavaScript 文件的压缩做额外的配置。

作为另一个受欢迎的 Web 服务器，Nginx 也内置了 `gzip` 作为 HTTP 压缩。压缩对 JavaScript、HTML、CSS 和 XML 文件也是默认启用的。如果你在使 Nginx，你也不需要针对 JavaScript 文件的压缩做额外的配置。

IE 6 和早期的浏览器在特定情况下处理 HTTP 压缩有已知问题。Apache 2 和 Nginx 都允许你在必要的情况下针对这些浏览器关闭 HTTP 压缩。自从微软在 2012 年开始自动把 Internet Explorer 6 升级至 Internet Explorer 8 之后，这应该不再是问题了。

17.2.2 构建时压缩

你可能希望在构建时自己压缩文件而不借助服务端的干预。`jQuery` 为主要的 JavaScript 文件构建了一个 `gzip` 压缩后的版本，可以从 <http://jquery.com> 下载。在构建期借助 Ant 的`<gzip>`任务可以很容易地执行 Gzip 压缩。

<gzip>任务仅仅适用于单个文件。你需要通过 `src` 属性指定要压缩文件的文件名和 `destfile` 属性指定输出的文件。例如：

```
<gzip src="${build.dir}/build.js" destfile="${build.dir}/build.js.gz"/>
```

该任务对 `build.js` 执行了 `gzip` 压缩并输出结果到 `build.js.gz`。如果你只有一个文件需要压缩，当然可以通过这种方式使用。例如：

```
<target name="compress">  
    <gzip src="${build.dir}/build.js" destfile="${build.dir}/build.js.gz"/>  
</target>
```

这种情况下，文件名应当存储在属性文件中以便后续可以方便地更改。

通过 `gzip` 压缩多个文件而不单独设置它们的文件名需要一点小创新。Ant 本身并没有提供循环遍历一个文件列表的方法。然而，可以使用 Ant 内嵌的 JavaScript 提供这个方法。

Ant 中的<script>任务允许你使用多种语言书写脚本，JavaScript 只是其中之一。通过设置 `language` 属性为“`JavaScript`”可以指定 JavaScript 为你当前使用的语言。紧接着，通过 `CData` 将<script>元素的内容包裹起来，如下所示。

```
<script language="javascript"><![CDATA[  
    // 这里写代码  
]]></script>
```

添加 `CData` 分隔符可以确保你无需担心 `script` 内字符的转义。

<script>任务内部的 JavaScript 代码在类似 Rhino 默认环境的环境下执行。你可以访问 `Java` 对象，也可以通过 `importPackage()` 函数导入更多的扩展。也有一个 `project` 对象指向 `build.xml` 中定义的全局项目。你可以通过 `project.getProperty()` 方法读取属性，通过 `project.createTask()` 方法创建新的任务。把这些片段拼起来，就可以创建一个压缩构建目录下所有文件的 `compress` 目标。

```
<target name="compress">
```

```

<!-- 将文件名存储为属性，使用 ; 来做分界 -->
<pathconvert pathsep=";" property="compress.jsfiles">
    <fileset dir="${build.dir}" includes="*.js"/>
</pathconvert>

<script language="javascript"><![CDATA[
    importPackage(java.io);

    <!-- 得到属性并将其转换为数组 -->
    var files = project.getProperty("compress.jsfiles").split(";");
    gzip,
    i,
    len;

    for (i=0, len=files.length; i < len; i++) {
        // 创建一个 gzip 任务
        gzip = project.createTask("gzip");
        gzip.setSrc(new File(files[i]));
        gzip.setDestfile(new File(files[i].replace(".js", ".js.gz")));
        gzip.perform();
    }

    ]]> </script>
</target>

```

`compress` 目标的第一部分将`<fileset>`转换为属性。属性 `compress.jsfiles` 填充为一个用分号分隔的文件名列表。在`<script>`任务内部，第一行导入了 `java.io` 包，因此便可以使用 `File` 类了。接下来，`compressor.jsfiles` 属性将被读取并且用分号分离开，因此 `files` 是一个文件名列表的数组。

而后，`for` 循环用来遍历这些文件名。对于每个文件名，都会通过 `project.createTask("gzip")` 来创建一个新的`<gzip>`任务。`gzip` 变量包含一个代表该任务的 Java 对象。每个属性都拥有设置其值和取得其值的方法，`setSrc()` 用来设置 `src` 属性，`setDestfile()` 用来设置 `destfile` 属性。这些属性都代表文件，因此需要传递 `File` 的实例而不是文件名。输出文件通过文件名的 `replace()` 方法被赋值为 `.js.gz` 扩展名。最后一步是调用真正执行任务的 `perform()` 方法。

这个版本的 `compress` 目标不需要事先已知文件名，因此更加通用。

Buildr 的`<gzipall>`任务也可以方便地压缩多个文件。

```
<target name="compress">
  <gzipall>
    <fileset dir="${build.dir}" includes="*-min.js" />
  </gzipall>
</target>
```

使用<gzipall>会自动给 gzip 压缩后的文件名添加.gz 后缀。

如果你倾向于手动压缩这些文件，依旧需要给服务器做配置发送 Content-Encoding: gzip 头信息以便浏览器可以正常地使用这些压缩后的文件。



文档化

所有工程师都喜欢写代码，不喜欢写文档。这也恰恰说明为什么自动从代码生成文档的工具如此受欢迎。这一趋势始于 Javadoc，它会自动为 Java 创建文档，而后延伸到了其他语言，如 JavaScript。

当前，针对 JavaScript 的文档生成工具越来越多^①。有些是通用的适用于任何语言的文档的生成工具；有些则是专门针对 JavaScript 的。正如压缩工具一样，文档生成工具的选择更多的是一种个人偏好而非其他原因。这一章介绍了一些流行的文档生成工具。除此之外，更多的替代选择请参阅附录 B。

18.1 JSDoc Toolkit

JSDoc Toolkit 可能是最常用到的 JavaScript 文档生成工具了。它作为一个原始 JSDoc 的改进版本在 2011 年发布，已经被 Google 和 SproutCore 所使用并一度被认为是引领了 JavaScript 专用文档生成器的趋势。它采用了同 Javadoc 相同的基本语法，通过特殊的多行注释指定文档信息。例如：

```
/**  
 * @namespace 应用程序单体.  
 */  
var MyApplication = {  
    /**
```

^① 译注：这些文档生成工具大多有一个通病，就是对源文件编码类型“很挑剔”，如果你的项目中 js 文件既有 gbk 编码的文件，又有 utf-8 编码的文件，生成的文档就会有乱码，译者实现了一个 JSDoc 的辅助脚本，来对这种混乱编码的情况进行纠错：<https://github.com/jayli/jsdoc-packer>。

```
* 两个数字的加法
* @param {int} num1 第一个数字.
* @param {int} num2 第二个数字.
* @returns {int} 两个数的和.
* @static
*/
add: function (num1, num2) {
    return num1 + num2;
}
}
```

在 JSDoc 中任何没有可调用构造函数的对象都当做命名空间，因此 `MyApplication` 就是一个如`@namespace` 标签后所描述的一个命名空间。`MyApplication.add()`方法接收两个参数，`@param` 后边跟着预期的数据类型、参数名和描述信息。方法会有一个返回值，所以`@return` 标签表示了返回值预期的数据类型和描述。`@static` 标签表示该方法无需作为对象实例运行。

JSDoc Toolkit 处理 JavaScript 文件的过程中，会根据所有 JavaScript 代码中的文档注释来生成基于 HTML 格式的文档。更详尽的语法信息参见：<http://code.google.com/p/jsdoc-toolkit/w/list>。

JSDoc Toolkit 几乎完全由 JavaScript 语言打造，并且采用一个定制的 Rhino 启动器的 JAR 文件 (`jsrun.jar`) 来执行。

```
java -jar jsrun.jar app/run.js [file]+ -t=[templates] -d=[directory] [options]
```

`app/run.js` 文件是 JSDoc Toolkit 主要的可执行文件。你可以指定任意想要生成文档的 JavaScript 文件。`-t` 标记指定了所要采用的模板（默认可以使用 JSDoc Toolkit 自带的模板），`-d` 标记指定了输出目录。例如：

```
java -jar jsrun.jar app/run.js core/core.js -t=templates/jsdoc/ -d=./out
```

这条命令采用 `templates/jsdoc/` 作为模板，为 `core/core.js` 生成文档，并且输出最终的 HTML 文档到 `out` 目录。因为你可能会多次用到 JSDoc Toolkit 目录 (`jsrun.jar`, `app/run.js`, 以及默认的模板都有可能用到)，所以最好将这个信息存放在属性中，如下所示。

```
src.dir = ./src
lib.dir = ./lib
```

```
jsdoc.dir = ${lib.dir}/jsdoc-toolkit
jsdoc = ${jsdoc.dir}/jsrun.jar
jsdoc.run = ${jsdoc.dir}/app/run.js
jsdoc.templates = ${jsdoc.dir}/templates
jsdoc.output = ./docs
```

一个产生文档的目标如下。

```
<target name="document">
    <apply executable="java" failonerror="true" parallel="true">
        <fileset dir="${src.dir}" includes="**/*.js" />
        <arg line="-jar"/>
        <arg path="${jsdoc}"/>
        <arg path="${jsdoc.run}" />
        <arg line="-t=${jsdoc.templates}" />
        <arg line="-d=${jsdoc.output}" />
        <srcfile/>
    </apply>
</target>
```

因为构建后的文件会去除所有注释，所以 `document` 目标会根据源文件生成文档。同 `validate` 目标类似，将 `<apply>` 任务的 `parallel` 属性置为 “`true`” 可以使所有的文件在命令行上传递一次。剩下的只是 `<arg>` 元素指定不同的选项。默认情况下，文档会生成到顶层的 `docs` 目录，但是你可能需要根据你的目录结构对这个位置做出适当的调整。

Buildr 任务在 JSDoc 中的标签是 `<JSDoc>`，`outputdir` 是它不可或缺的属性。除此之外还有一个可选的 `templates` 属性（如果你不想使用默认的模板）。例如：

```
<target name="document">
    <jsdoc outputdir="${jsdoc.output}">
        <fileset dir="${src.dir}" includes="**/*.js" />
    </jsdoc>
</target>
```

这个目标同上个写法做着同样的事儿，但是很显然在代码的解释性方面要比上边清晰得多。

18.2 YUI Doc

最初版本的 YUI Doc 是用 Python 写的并且被 YUI library 使用了几个年头。最近，一

一个新的 JavaScript 版本被创造出来了，采用相同的语法。这是一个在 <http://yuilibrary.com> 上的生成文档的工具。因为它继承了 Javadoc 风格的注释，所以其语法同 JSDoc 相近。最大的区别就是 YUI Doc 需要你在文档注释中命名你的属性和方法，而 JSDoc 能够分析 JavaScript 代码从而推断出这个名字。

举个例子。

```
/**  
 * The main application object.  
 * @class MyApplication  
 * @static  
 */  
var MyApplication = {  
    /**  
     * 两个数字的加法。  
     * @param {int} num1 第一个数字。  
     * @param {int} num2 第二个数字。  
     * @returns {int} 两个数的和。  
     * @method add  
    */  
    add: function (num1, num2) {  
        return num1 + num2;  
    }  
}
```

就 YUI Doc 而言，MyApplication 即便没有构造函数也会被认为是一个类。第一行是类的描述，`@class MyApplication` 表明该对象作为一个类。因为没有构造函数，`@static` 标签表明 MyApplication 是一个对象，并且它的所有方法均可以被静态地访问。这个 `MyApplication.add()` 方法语法非常类似于 JSDoc，主要的区别就在于`@method` 标签指明该方法的名称。

YUI Doc 是用 JavaScript 语言编写的并且运行在 Node.js 环境上。可以通过 npm 安装它。

```
sudo npm install -g yuidoc
```

YUI Doc 的命令行看起要比 JSDoc 简单些。

```
yuidoc [options] [directory]+ -o [directory]
```

例如：

```
yuidoc ./src -o ./docs
```

这条命令会递归地查找 `src` 目录并且解析所有它找到的 JavaScript 文件。最终会在 `docs` 目录下生成 HTML 格式的文档。尽管 YUI Doc 有一些命令行参数，但是没有必要为了启动和运行而设置它们。YUI Doc 的属性很简单。

```
src.dir = ./src
lib.dir = ./lib
yuidoc = yuidoc
yuidoc.output = ./docs
```

你也许会质疑 `yuidoc = yuidoc` 的作用是什么，因为它看起来似乎是多余的。最好始终把应用程序的路径作为属性，因为路径和文件名有随时间变化的倾向。尽管这些在现在都是相同的，但是没有人知道未来是怎样的。当发生改变的时候，你更希望能够在属性文件中快速做出相应的改变而不是到 `build.xml` 文件中来找出可执行文件的设置。

因为 YUI Doc 可以指定一个或多个目录而不是文件，所以目标就变得非常简单，其中只有一个源目录。

```
<target name="document">
  <exec executable="yuidoc" failonerror="true">
    <arg path="${src.dir}" />
    <arg line="-o" />
    <arg path="${yuidoc.output}" />
  </exec>
</target>
```

这个目标使用`<exec>`代替了`<apply>`，因为只有一个目录需要处理。`<exec>`任务除了不需要`<srcfile>`元素外其余同`<apply>`功能类似，并且同`<arg>`元素构成了完整的命令行。

Buildr 中的`<yuidoc>`任务对这个功能做了封装，有两个必需的属性，`inputdir` 和 `outputdir`，分别指定在 JavaScript 文件的目录路径和生成文档的路径。例如：

```
<target name="document">
  <yuidoc inputdir="${src.dir}" outputdir="${yuidoc.output}" />
</target>
```

`<yuidoc>`任务使用之前需要确保你已经安装了 YUI Doc。

第 19 章

自动化测试

JavaScript 的测试一直以来都是开发人员的心病^①。你期望简单快速地测试 JavaScript，但是又有众多的浏览器需要测试。第一个解决方案就是人工的跨浏览器测试，这将意味着创建一个 HTML 文件并且手动地在众多的浏览器中加载并确保它正常工作。尽管可行，但这种方法在实际使用中效率太低。

JavaScript 测试的下一波浪潮聚集在了根植浏览器环境的命令行测试。通过 Rhino 和模拟的浏览器环境在命令行上也尝试了 JavaScript 测试。有些公司甚至开发出了浏览器外观并声称能够在跨浏览器测试中加载。遗憾的是这些植入的浏览器环境并没有做这个工作。试图重建一个真正独一无二的手工环境存在结果不一致的风险：你的测试可能会在“假 Firefox”中通过，但是在实际的浏览器中却可能会失败。

最近，在真实的浏览器环境中测试做了一些尝试。

这种方法通常需要引入一个 HTML 文件来启动测试，然后通过一个应用程序在不同的浏览器中加载该文件。本章中提到的许多工具都还处于开发阶段，但是它们都能让你对集成的基于浏览器的 JavaScript 测试有一个很好的起点。

19.1 YUI Test Selenium 引擎

YUI Test (<http://yuilibrary.com/projects/yuitest>) 是 YUI Library 中的一个测试框架。

^① 译注：前端测试的话题由来已久，关于一些背景和难题的介绍请阅读《JavaScript Web 富应用开发》第 9 章“测试和调试”。

最新版本的 YUI Test 不仅仅是一个简单的测试库。

除了去除了对核心 YUI 库的依赖之外，YUI Test 还支持一套实用工具来辅助 JavaScript 测试。这套工具叫做 YUI Test Selenium 引擎，设计用来同 Selenium 一起工作，同时又能容易地进行浏览器测试。

Selenium (<http://seleniumhq.com>) 是一个能够启动浏览器并在它们内部执行命令的服务器。原本设计用于 QA 工程师编写功能测试的，由于 Selenium 能方便地与浏览器交互而在 JavaScript 测试中流行起来了。

19.1.1 配置一台 Selenium 服务器

YUI Test Selenium 引擎同 Selenium 服务器一同在众多的浏览器中执行 JavaScript 测试用例并返回相应的结果。第一步就是配置一台 Selenium 服务器（如果先前没有的话）。Selenium 是 Java 语言写的，所以它可以运行在任何安装有 Java 环境的地方。从 <http://seleniumhq.org/download/> 可以下载到最新的 Selenium 版本。

若要启动 Selenium 服务，可以到包含下载到的文件的目录下运行。

```
java -jar selenium-server-standalone-x.y.z.jar
```

服务的启动需要花一段时间，而后就可以接收我们发出的指令了。

19.1.2 配置 YUI Test Selenium 引擎

配置 YUI Test Selenium 引擎需要三个步骤。

1. 下载最新版本的 YUI Test。
2. 将 `yuitest-selenium-driver.jar` 放置在你的依赖目录中。
3. 把 `selenium-java-client-driver.jar` 从 YUI Test 的 lib 目录拷贝到你的 Java 运行环境 (JRE) 的 lib/ext 目录中。

当这三步完成后，现在就可以通过 YUI Test Selenium 引擎执行测试了。

19.1.3 使用 YUI Test Selenium 引擎

YUI Test Selenium 引擎通过 HTML 文件来执行测试，即便你的测试是在独立的

JavaScript 文件中的，你也需要在 HTML 文件中引入来在页面载入的时候自动执行测试。下面是一个测试页面的例子。

```
<!DOCTYPE html>
<html>
<head>
    <title>YUI Test</title>

    <!-- 引入 YUI Test 库 -->
    <script src="yuitest.js"></script>

    <!-- 引入测试文件 -->
    <script src="tests1.js"></script>
    <script src="tests2.js"></script>
</head>
<body>
    <script>
        YUITest.TestRunner.run();
    </script>
</body>
</html>
```

每一个 JavaScript 测试文件中都应该使用 YUITest.TestRunner.add()添加测试用例。

倘若这个 HTML 文件存放在服务器上，诸如 <http://www.example.com/test/html>，你可以通过下述命令执行测试。

```
java -jar yuitest-selenium-driver.jar [options] [url]+
```

例如：

```
.java -jar yuitest-selenium-driver.jar http://www.example.com/tests.html
```

这条命令将在 Firefox (Selenium 中的默认浏览器) 中执行所给文件，前提是 Selenium 服务已经运行在 localhost:4444 (Selenium 默认端口)。你也可以通过附加 options 来改变 Selenium 的位置。

```
java -jar yuitest-selenium-driver.jar --host testing.example.com
--port 9000 http://www.example.com/tests.html
```

这条命令将通过 testing.example.com:9000 的 Selenium 服务在 Firefox 中执行所给文

件。你同样可以用 Selenium IDs 指定额外的浏览器。

```
java -jar yuitest-selenium-driver.jar  
--browsers *firefox,*iexplore http://www.example.com/tests.html
```

这条命令将在 Firefox 和 Internet Explorer 中执行测试。--browsers 选项直接将这些参数传递给 Selenium，因此必须制定一个有效的 Selenium 浏览器。如果所给浏览器在指定的 Selenium 服务器中不是一个有效的浏览器就会报错。

即便可以在命令行指定测试 URL，但是大多数开发者还是使用 XML 配置文件来指定。XML 文件的格式如下。

```
<?xml version="1.0"?>  
<yuitest>  
  <tests base="http://www.example.com/tests/" timeout="10000">  
    <url>test_core</url>  
    <url timeout="30000">test_util</url>  
    <url>test_ui</url>  
  </tests>  
</yuitest>
```

<test>元素用来指定相对根路径，和每个测试的默认超时时间。接着，<url>元素指定了测试页面的相对路径。通过--tests 参数可以告诉 YUI Test Selenium Driver 使用 XML 文件代替命令行参数。

```
java -jar yuitest-selenium-driver.jar --tests tests.xml
```

于是 YUI Test Selenium Driver 就会在指定的浏览器中遍历执行这些测试，并返回所有的结果。

--erroronfail 参数用来指明当测试失败时候 YUI Test Selenium Driver 退出并返回非零状态码。指明这个参数是一个好主意，这样当测试失败时构建就会停止。

19.1.4 Ant 的配置写法

要创建一个 Ant 目标（Ant target），开始需要在属性文件中指定关键的一些数据。

```
src.dir = ./src  
lib.dir = ./lib  
tests.dir = ./tests
```

```
yuitestselenium = ${lib.dir}/yuitest-selenium-driver.jar  
yuitestselenium.host = testing.example.com  
yuitestselenium.port = 4444  
yuitestselenium.tests = ${tests.dir}/tests.xml  
yuitestselenium.browsers = *firefox
```

Ant 目标使用<exec>任务来执行 YUI Test Selenium Driver 并且传递相关信息。

```
<target name="test">  
  
<exec executable="java" failonerror="true">  
    <arg line="-jar"/>  
    <arg path="${yuitestselenium}"/>  
    <arg line="--host ${yuitestselenium.host}"/>  
    <arg line="--port ${yuitestselenium.port}"/>  
    <arg line="--browsers ${yuitestselenium.browsers}"/>  
    <arg line="--tests ${yuitestselenium.tests}"/>  
    <arg line="--erroronfail"/>  
</exec>  
  
</target>
```

如若 Selenium 服务没有运行或者指定的浏览器不存在，测试目标都会执行失败。在执行测试之前检查所有这些前置条件是很重要的。

Buildr 提供的<yuitest-selenuim>任务封装了所有这些功能，其中 onfail 标志始终会作为参数传递，其他选项可以作为属性。

```
<target name="test">  
  
<yuitest-selenium host="${yuitestselenium.host}"  
    port="${yuitestselenium.port}" browsers="${yuitestselenium.browsers}"  
    tests="${yuitestselenium.tests}"/>  
  
</target>
```

对于<yuitest-selenium>而言，tests 属性是必须的，其他属性都是可选的。

19.2 Yeti

Yeti (<http://yuilibrary.com/projects/yeti>) 是另一个设计来与 YUI Test 一起工作的工

具。与 YUI Test Selenium Driver 不同的是，Yeti 是一个完全独立的解决方案，用 JavaScript 写测试并在 Node.js 上运行。Yeti 需要你有一个 HTML 文件来自动执行你的测试，因此你可以使用在 YUI Test Selenium Driver 中相同的 HTML 文件。

你可以通过 npm 安装 Yeti。

```
sudo npm install -g yeti
```

运行 Yeti 仅仅需要简单地在命令行把你的 HTML 文件作为参数即可。

```
yeti test.html
```

这个命令将默认在 Firefox 或者 Safari（视平台而定）中执行所有的测试用例并且输出结果到命令行。若 Yeti 服务是在本地运行的，该命令也会在所有关联的浏览器中执行测试并汇报所有结果。

由于 Yeti 的简便，所以 Ant 目标中只需要在属性文件中做很少的配置。

```
src.dir = ./src
lib.dir = ./lib
tests.dir = ./tests

yeti = yeti
```

Ant 目标本身也是非常简单的。

```
<target name="test">

    <apply executable="yeti" failonerror="true" parallel="true">
        <fileset dir="${tests.dir}" includes="**/*.html" />
        <srcfile/>
    </apply>

</target>
```

test 目标通过<apply>任务来将所有在 tests 目录下找到的 HTML 文件传递给 Yeti。结果将输出到屏幕上，并且遇到错误的时候会构建失败。

Buildr 中的<yeti>任务可以让 Yeti 的使用更简便。

```
<target name="test">  
    <yeti>  
        <fileset dir="${tests.dir}" includes="**/*.html" />  
    </yeti>  
</target>
```

需要注意的是<yeti>任务需要在执行构建脚本的计算机上事先安装好 Yeti。

19.3 PhantomJS

Wekit 是驱动 Safari 和 Chrome 的渲染引擎。PhantomJS (<http://www.phantomjs.org>) 就是一个命令行版本的 Webkit，尽管并非完全相同，但是它的表现跟这些浏览器十分相近，并且可以不打开浏览器就能执行真正的浏览器测试。PhantomJS 附带的脚本可以在两个不同的测试框架中运行测试：Jasmine 和 QUnit。

PhantomJS 不仅仅是一个浏览器——它也是一个浏览器的脚本环境。PhantomJS 附带的一部分脚本可以运行 Jasmine 和 QUnit。这些脚本都需要使用适用于框架的 HTML 页面模板。

19.3.1 安装及使用

如果你用的是 Ubuntu，你可以通过 apt-get 安装。

```
$ sudo add-apt-repository ppa:jerome-etienne/neoip  
$ sudo apt-get update  
$ sudo apt-get install phantomjs
```

如果你在 Mac OS X 上使用 Homebrew，可以通过下面的命令安装 PhantomJS。

```
brew install phantomjs
```

其他的平台可以从 <http://code.google.com/p/phantomjs/downloads/list> 下载适用于你的操作系统平台的最新可执行文件。将整个 PhantomJS 目录放在一个便于访问的位置（你的依赖目录或者其他地方）。运行 Jasmine 和 QUnit 测试的脚本文件在 examples 目录下。

如果你是用 Homebrew 或者 apt-get 安装的，你需要单独从 PhantomJS 仓库下载运

行 Jasmine 和 QUnit 的文件，因为这些默认是不包含的。

PhantomJS 在命令行通过如下格式执行 Jasmine 和 QUnit 测试。

```
phantomjs [driver] [HTML file]
```

例如，运行 QUnit。

```
phantomjs example/run-qunit.js tests.html
```

运行 Jasmine。

```
phantomjs examples/run-jasmine.js tests.html
```

运行结果将输出在命令行。

19.3.2 Ant 的配置写法

像 Yeti 一样，PhantomJS 的 Ant 目标也相当简单。只有几个属性来跟踪。

```
src.dir = ./src
lib.dir = ./lib
tests.dir = ./tests

phantomjs = phantomjs
phantomjs.driver = ${lib.dir}/phantomjs/examples/run-qunit.js
phantomjs.tests = tests.html
```

因为 PhantomJS 的测试管理工具一次只支持传递一个文件，所以 Ant 目标使用<exec>代替<apply>。

```
<target name="test">

    <exec executable="phantomjs" failonerror="true">
        <arg path="${phantomjs.driver}" />
        <arg path="${tests.dir}/${phantomjs.tests}" />
    </exec>

</target>
```

test 目标仅仅传递两个路径给执行程序，且当遇到错误时执行失败。

Buildr 的<phantomjs>任务可以使你通过稍有不同的语法来执行同样的操作。

```
<target name="test">

    <phantomjs driver="\${phantomjs.driver}">
        <fileset dir="\${tests.dir}" includes="*.html" />
    </phantomjs>

</target>
```

driver 属性是必不可少的，用来指定用于你的测试的 PhantomJS 驱动器。<phantomjs> 任务使用一个或者多个<fileset>元素来指定要执行的测试。除此之外，它和前一个 test 目标的表现效果完全一致。

19.4 JsTestDriver

JsTestDriver (<http://code.google.com/p/js-test-driver/>) 是由 Google 工程师开发的与 Selenium 和 Yeti 类似的命令行工具。JsTestDriver 基于已安装的浏览器运行测试。JsTestDriver 也有自己的测试框架，所以默认情况下，你必须使用它提供的库来写测试。通过 QUnit 适配器可以用 JsTestDriver 来执行基于 QUnit 的测试，如果你愿意，也可以写自己的适配器。

19.4.1 安装及使用

JsTestDriver 是用 Java 编写的，所以你必须先在你的依赖目录下载最新的 JAR 文件。在开发机上手动将浏览器连接到 JsTestDriver 服务器是最常见的使用 JsTestDriver 的模式。YAML 文件中包含要执行测试的文件的配置信息大致如下。

```
server: http://localhost:4224

load:
  - tests/*.js
```

第一行表明 JsTestDriver 服务器所在的地址，load 部分指明了要加载的测试文件。为了方便执行构建，JsTestDriver 提供了一个自动启动和停止浏览器同时收集测试结果的命令。格式如下。

```
java -jar JsTestDriver.jar --port [port] --browser [browsers] --config
[file] --tests all --testOutput [directory]
```

例如：

```
java -jar JsTestDriver.jar --port 4224 --browser firefox,iexplore --config  
conf/conf.yml --tests all --testOutput ./results
```

这条命令将在 Firefox 和 Internet Explorer 中执行所有 conf/conf.yml 中指定的测试并输出执行结果。--browsers 选项需要指定浏览器所在的路径，因此这个例子假定 Firefox 和 Internet Explorer 在不完整路径下是可执行的。

19.4.2 Ant 的配置写法

为 JsTestDriver 创建一个目标需要指定一些关键的信息片段。

```
src.dir = ./src  
lib.dir = ./lib  
tests.dir = ./tests  
  
jstestdriver = ${lib.dir}/JsTestDriver.jar  
jstestdriver.port = 4224  
jstestdriver.browser = firefox,iexplore  
jstestdriver.config = conf/conf.yml  
jstestdriver.output = ./results
```

其 Ant 配置写法看起来跟 YUI Test Selenium 引擎的写法十分类似。

```
<target name="test">  
  
<exec executable="java" failonerror="true">  
    <arg line="-jar"/>  
    <arg path="${jstestdriver}"/>  
    <arg line="--port ${jstestdriver.port}"/>  
    <arg line="--browser ${jstestdriver.browser}"/>  
    <arg line="--conf"/>  
    <arg path="${jstestdriver.config}"/>  
    <arg line="--tests all"/>  
    <arg line="--testOutput"/>  
    <arg path="${jstestdriver.output}"/>  
</exec>  
  
</target>
```

按照目前的配置，这个 Ant 目标将在 Firefox 和 Internet Explorer 中执行所有配置文件中指定的测试。Buildr 中的<jstestdriver>任务可以更简单地使用 JsTestDriver。有两个必备的属性：outputdir 指定输出结果的路径，config 指定配置文件的位置。其他所有命令行参数都可以作为属性指定。下述目标跟最后一个例子等同。

```
<target name="test">  
    <jstestdriver config="${jstestdriver.config}"  
        outputdir="${jstestdriver.output}"  
        tests="all" port="${jstestdriver.port}"  
        browser="${jstestdriver.browser}" />  
</target>
```

组装到一起

先前关于“构建系统”(build system)的章节都集中讲解如何创建系统的某一部分，创建一个实用的工具库，我们可以很轻松地将它们组合在一起。本章将着重介绍如何将“各个部分”组装成一个适用你 JavaScript 项目的完整的解决方案。你的系统最后可能有很多复杂的功能，也可能像 Buildr 一样把任务捆绑起来，但和别的“构建系统”(build system)一样，也应包含一些常见的功能块。

20.1 被忽略的细节

在组装“构建系统”之前，有一些容易遗漏的细微步骤。首先是 build 目录的创建。由于该目录是临时的（并且不会被提交），由构建系统负责生成。Ant 中的<mkdir>任务可以很方便地实现。

```
<target name="init">
  <mkdir dir="${build.dir}" />
</target>
```

init 目标只做了一件事：创建一个构建目录以便存放所有构建后的文件。不过诸如<concat>这样的任务也可能最终创建该目录，如果事先它并不存在的话。然而，最好显式地声明每一步的构建过程，以确保任务或者目标的重新排序不会导致错误。

其次清理 build 目录也是容易疏漏的部分。在构建过程中，如果想要删除所有的文件并且从头开始。最快的方法就是使用<delete>任务简单地删除 build 目录。

```
<target name="clean">
  <delete dir="${build.dir}"/>
</target>
```

clean 操作将会删除 build 目录，因此你肯定会得到最新的文件。

20.2 编制打包计划

把构建系统的各个部分按照正确的顺序组合在一起，可以帮助你想清楚开发过程，和你需要的不同的构建类型。鲜有一个项目只用到一种构建类型，至少有三种常见的。

开发

开发版本是由开发人员驱动的，因为他们的工作需要基于此。这个版本的构建时间应当尽可能的快以便不会中断开发者的开发过程。一般而言，你希望阻止意外的错误并且保持良好的测试状态，这样开发者就能在浏览器加载程序时手动执行测试。这个构建过程不应当超过 15 秒，所以你需要仔细选择你要做的事情。如果你在做一个 Web 应用，你可能希望有一个独立于整个 Web 应用之外的独立的 JavaScript 开发版本。这通常也是默认的构建类型。

集成

集成版本是一个按照固定的时间表自动构建的版本。有些时候每一次提交会自动构建，但是在大型项目中，它们往往是每个几分钟运行一次。集成版本负责寻找整个系统中的问题。由于它是自动化的，所以这个构建过程可以花费稍长的时间但是必须尽可能的详尽。在某些情况下，集成版本是把代码改动推向生产环境的最后一道防线。

发布

一个按需执行的构建，仅仅在推送到生产环境之前执行。这种构建的主要工作是获得代码的最终版本以便部署。理论上讲，如果集成的操作按规定完成，那么发布版本应当是无误的。但是实际情况并非总是如此，发布版本可能会承担更多的任务，也会出现一些部署前的错误。

当然，你的项目在你的开发过程中可能产生很多构建类型。

你的 build.xml 文件开头看起来应该是像下面这样的。

```
<project name="yourapp" default="build.dev">

    <!-- 引入属性文件 -->
    <loadproperties srcfile="yourapp.properties" />

    <!-- 在这里定义或引入实用目标 -->

    <!-- 初始化和清理 -->
    <target name="init">
        <mkdir dir="${build.dir}" />
    </target>

    <target name="clean">
        <delete dir="${build.dir}" />
    </target>

    <!-- 主构建 -->
    <target name="build.dev">
    </target>

    <target name="build.int">
    </target>

    <target name="build.release">
    </target>

</project>
```

首先确保已经导入了属性文件以便构建目标获取所有需要的数据。接下来，包含或者导入像先前章节所创建的实用目标。有时候这么做有助于保持实用目标在一个或多个独立的 XML 文件，但是最终还是取决于你。而后 init 和 clean 目标完成了实用目标。最后一节包含了生成每一种构建类型的目标。

20.2.1 开发版本的构建

确保开发版本的正确性是很重要的，因为它将直接关系到每一个开发人员的工作流程。一般而言，开发版本旨在让代码部署到开发环境尽可能的快，然而依然需要执行一些完整性检查。大多数开发版本都仅做两件事：验证代码和连接文件。因为开发人员调试需要完整的源代码，因此就这一点而言压缩代码是毫无意义的。`build.dev` 目标大致如下。

```

<project name="yourapp" default="build.dev">

    <!-- omitted for clarity -->

    <!-- main builds -->
    <target name="build.dev" depends="clean,init,validate,concatenate">
    </target>

</project>

```

所有主要的构建目标都应当看起来如此简单，因为它们仅仅是将其他目标用特定的顺序捆绑在一起。`build.dev` 目标指明了自身需要依赖 `clean`、`init`、`validate` 和 `concatenate` 目标。因此执行 `ant build.dev` 将仅执行上述四个目标而已。如前所述，你应当始终保持开始全新的构建，因此首先确保已经删除了旧的文件。因为 `concatenate` 目标也会将构建文件放置在正确的目录，当所有的代码执行完毕后 `build.dev` 目标就宣告完成了。

开发版本是可选的，但是亦可能想要在开发版本中包含一些测试或者让开发者单独执行 `ant test`。不要忘了因为要配置和卸载浏览器，所以测试需要话费更多的时间。

20.2.2 集成版本的构建

集成版本往往作为持续集成（CI）系统^①的一部分而自动运行，因此它需要更多步骤来完成。因为该版本是主要防御错误的措施，所以它应当包含尽可能多的校验和测试。至少，它应当包含所有开发版本做的工作以及单元测试。而且这也是一个放置文档生成的绝佳选择，以便其他开发人员清楚地看到代码究竟做了哪些改动。

这里有一个例子。

```

<project name="yourapp" default="build.dev">

    <!-- omitted for clarity -->

    <!-- main builds -->
    <target name="build.int" depends="build.dev,minify,test,document">
    </target>

</project>

```

^① 译注：持续集成简称 CI，持续集成是频繁、持续地在多个团队成员的工作中进行集成，并且给与反馈。

`build.int` 目标首先执行 `build.dev` 完成校验和代码合并。紧接着，代码被压缩、测试、文档化。如果该构建过程被中断，它将导致 CI 系统通过适当的途径报告问题。有的系统会发送电邮，有的会在稳定的公告版上表明可视的变化。任何情况下，集成版本的构建中断都需要迅速解决以免其他开发人员不会因此而阻塞。

你可能希望在自动构建的过程中加入文档生成。原因在于未能生成文档并不一定意味着代码有问题。也可能是文档生成工具自身的原因。再者，这很大程度上依赖于你的开发进度和你认为什么是足够重要值得停下来去修复的。

20.2.3 发布版本的构建

发布版本是开发过程的终结。正是这个版本确保了代码适合生产环境。当代码到发布版本的时候，应当是被验证和测试过的，自动和手动都应做好准备。某些情况下，发布版本唯一要做的仅仅是加工文件，插入版权，版本号和其他相关的元信息。例如：

```
<project name="yourapp" default="build.dev">

    <!-- omitted for clarity -->

    <!-- main builds -->
    <target name="build.release" depends="build.int,bake">
        </target>

    </project>
```

`build.release` 目标只是简单地执行了集成构建和加工文件而已。你可能还希望发布版本能处理部署文件到服务器。对于 JavaScript 而言，这通常作为更大的 Web 应用或项目的部署构建过程的一部分，而非发布版本的一部分。然而，你当然可以包含一个上传文件到服务器的任务或者执行其他分配的任务。

另一个发布版本的选择是简单地从集成版本中拷贝过来然后在部署前执行加工。假设集成版本没有错误，这样做可以节省时间，但是需要确保你在部署完全相同的测试过的代码。再者，这个选择是针对具体项目的，因此记得同你的团队事先讨论。

20.3 使用 CI 系统

在创建一个可维护的项目中，使用构建系统仅仅是第一步，接下来是把构建系统集成到一个 CI 系统中。CI 系统是建立在某些操作或者定期间隔的基础上自动运行生成的。例如，你可能每小时构建一次来把所有的最新更新的文件部署到集成环境中。如果失败的话，它可以给开发者发送电子邮件，让他们来解决问题。在任何一个优秀的软件项目中，CI 系统都是非常 important 的一部分。好在针对 CI 系统有一些优秀的免费资源。

20.3.1 Jenkins

Jenkins 是目前最广泛使用的 CI 系统之一。这是一个基于 Java 的 Web 应用程序，用于多个版本的管理。默认情况下，它集成了多个源代码控制库，通过可扩展的插件库，几乎可以支持所有的版本。Jenkins 原生的支持 Ant 和 Shell 脚本，也就意味着通过 Jenkins 你可以重复使用任何现有的构建脚本。

设置 Jenkins 是很容易的，只需要下载最新的 WAR 文件，并启动它。

```
java -jar jenkins.war
```

Jenkins 启动的 Web 服务器可以通过 <http://localhost:8080> 访问。在你的 Web 浏览器中浏览到该位置，就可以准备好开始构建工作了。这项工作是把要执行的构建步骤集合起来。单击“New Job”链接来创建你的第一份工作。下一个页面，要求你选择你想要创建哪种类型的工作。如果在你的构建系统中只使用了 Ant，那么选择“Build a free-style software project”。创建新的工作后，会进入到配置页。

在配置页上有一些基本的选项，但真正有趣的部分在页面的下方，名为“构建触发器”。这部分决定构建执行的频率。在另一个构建完成后，或者基于时间程序，或者有代码提交到源代码控制系统中，它都可以触发。

对于你主要的集成版本，你可能更希望文件提交后执行构建，这种情况下最后一个选项很适合你。“Poll SCM”是指 Jenkins 将会轮询你的源代码控制系统（如前面小节指定的）。轮询的格式跟在 Linux 下设置一个时间任务如出一辙，因此@hourly 代表每小时检查代码控制系统（参见图 20-1）。

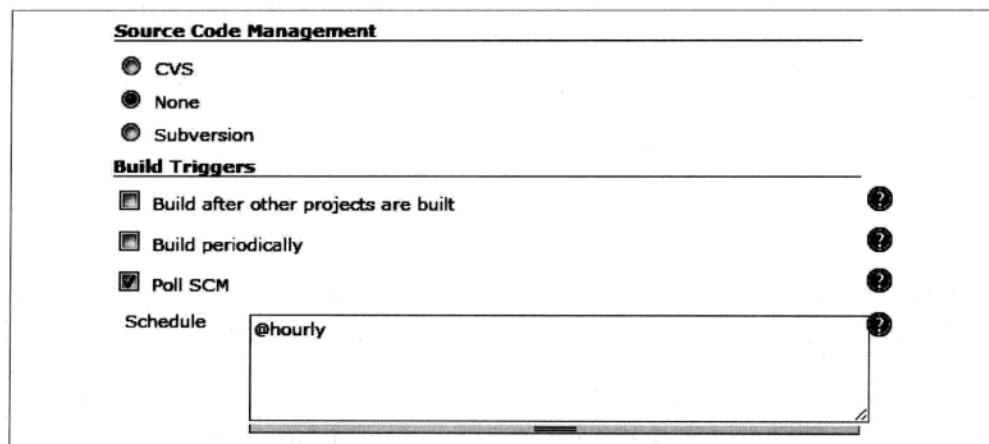


图 20-1 Jenkins 构建触发器

接下来，你需要设置一个任务来执行一个或者多个的 Ant 目标。为此，单击“Add Build Step”按钮会显示一个下拉菜单。选择“Invoke Ant”。页面上就会出现一个新的构建步骤，询问你指定要执行的 Ant 目标（参见图 20-2）。

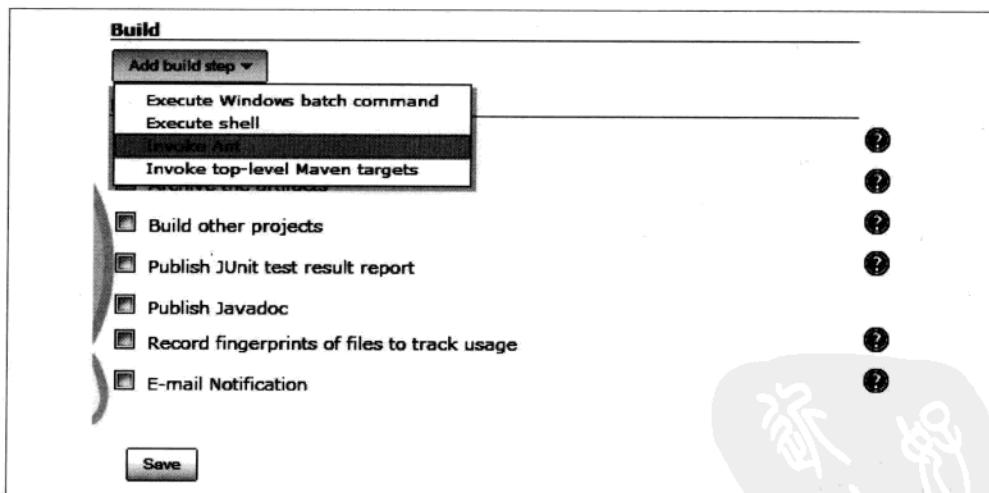


图 20-2 添加 Ant 构建步骤

如果你用的是 Jenkins 提供的源代码控制系统，它会在你的根目录自动找到 build.xml 文件，因此你只需要指定 Ant 目标的名称即可。如果你在设置集成版本的构建，例如，在文本框中输入 build.init 然后回车即可（参见图 20-3）。

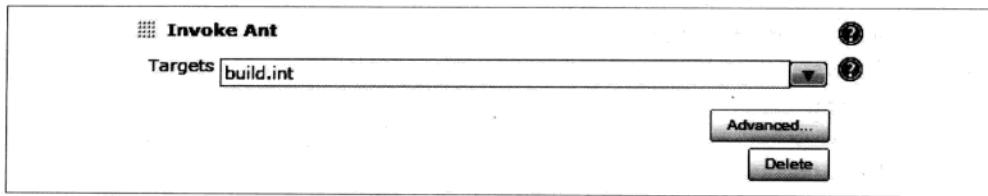


图 20-3 指定一个 Ant 目标

如果你在项目根目录中没有使用 build.xml，或者没有用 Jenkins 自带的源代码控制系统，需要单击“Advanced”按钮手动指定 build.xml 文件的路径。

而后，你可以指定构建失败时候发送邮件。Jenkins 允许你指定的邮箱始终接收该通知，同样可以发送邮件给中断构建的提交者（使用源代码控制系统检索电子邮件地址）。构建不成功时始终发送邮件是个好主意（参见图 20-4）。

Post-build Actions

- Aggregate downstream test results
- Archive the artifacts
- Build other projects
- Publish JUnit test result report
- Publish Javadoc
- Record fingerprints of files to track usage
- E-mail Notification

Recipients

Whitespace-separated list of recipient addresses. May reference build parameters like \$PARAM. E-mail will be sent when a build fails, becomes unstable or returns to stable.

Send e-mail for every unstable build

Send separate e-mails to individuals who broke the build

Save

图 20-4 配置邮件提醒

一旦你完成了邮件通知的设置，单击页面底部的“Save”保存构建任务。你的构建任务就会从即刻开始每小时执行一次。然而，你也可以到 Jenkins 点击“Build Now”来随时随地地手动执行构建。

这里仅仅简单介绍了 Jenkins 和可持续集成的功能。还有诸多的事情可以通过

Jenkins 来做，诸如跟踪单元测试的结果，发布构建日志，设置依赖构建等。Jenkins 网站是一个学习众多设置的一个伟大的资源。

20.3.2 其他 CI 系统

CI 在研究领域是很受欢迎的，所以始终有新的解决方案出炉。这里有一些可以考虑的免费的 CI 系统。

Continuum

一个 Apache 的 CI 项目，用以同 Ant 和 Maven 协同工作。可以从 <http://continuum.apache.org> 获得源代码。

BuildBot

一个基于 Python 的针对工程师的构建系统。可以从 <http://trac.buildbot.net> 获取源代码。

Cruise Control

另一个基于 Java 的作为 Web 应用的构建系统。也有 Ruby 和.NET 接口。可以从 <http://cruisecontrol.sourceforge.net> 获得源代码。

Gradle

基于 Groovy 语言的 CI 集成构建系统，结合了 Ant 和 Maven 两者的优势，对于没有编程基础的人而言上手会有一些困难。可以从 <http://www.gradle.org> 获取源代码。

附录 A

JavaScript 编码风格指南

程序语言的编码风格指南对于一个长期维护的软件而言是非常重要的。本指南是基于“Java 语言编码规范”(Code Conventions for the Java Programming Language)和 Crockford 的 (JavaScript) 编程规范，同时结合我个人的经验和喜好做了一些改动。

A.1 缩进

每一行的层级由 4 个空格组成，避免使用制表符 (Tab) 进行缩进。

```
// 好的写法
if (true) {
    doSomething();
}
```

A.2 行的长度

每行长度不应该超过 80 个字符。如果一行多于 80 个字符，应当在一个运算符 (逗号，加号等) 后换行。下一行应当增加两级缩进 (8 个字符)。

```
//好的写法
doSomething(argument1, argument2, argument3, argument4,
            argument5);

// 不好的写法：第二行只有 4 个空格的缩进
doSomething(argument1, argument2, argument3, argument4,
            argument5);
```

```
// 不好的写法：在运算符之前换行  
doSomething(argument1, argument2, argument3, argument4  
, argument5);
```

A.3 原始值

字符串应当始终使用双引号（避免使用单引号）且保持一行。避免在字符串中使用斜线另起一行。

```
// 好的写法  
var name = "Nicholas";  
  
// 不好的写法：单引号  
var name = 'Nicholas';  
  
// 不好的写法：字符串结束之前换行  
var longString = "Here's the story, of a man \  
named Brady.";
```

数字应当使用十进制整数，科学计数法表示整数，十六进制整数，或者十进制浮点小数，小数点前后应当至少保留一位数字。避免使用八进制直接量。

```
// 好的写法  
var count = 10;  
  
// 好的写法  
var price = 10.0;  
var price = 10.00;  
  
// 好的写法  
var num = 0xA2;  
  
// 好的写法  
var num = 1e23;  
  
// 不好的写法：十进制数字以小数点结尾  
var price = 10.;  
  
// 不好的写法：十进制数字以小数点开头  
var price = .1;  
  
// 不好的写法：八进制 (base 8) 写法已废弃  
var num = 010;
```

特殊值 `null` 除了下述情况下应当避免使用。

- 用来初始化一个变量，这个变量可能被赋值为一个对象。
- 用来和一个已经初始化的变量比较，这个变量可以是也可以不是一个对象。
- 当函数的参数期望是对象时，被用作参数传入。
- 当函数的返回值期望是对象时，被用作返回值传出。

例如：

```
// 好的写法
var person = null;

// 好的写法
function getPerson() {
    if (condition) {
        return new Person("Nicholas");
    } else {
        return null;
    }
}

// 好的写法
var person = getPerson();
if (person !== null){
    doSomething();
}

// 不好的写法：和一个未被初始化的变量比较
var person;
if (person != null){
    doSomething();
}

// 不好的写法：通过测试判断某个参数是否被传递
function doSomething(arg1, arg2, arg3, arg4){
    if (arg4 != null){
        doSomethingElse();
    }
}
```

避免使用特殊值 `undefined`。判断一个变量是否定义应当使用 `typeof` 操作符。

```
// 好的写法
if (typeof variable == "undefined") {
    // do something
}

// 不好的写法：使用了 undefined 直接量
```

```
if (variable == undefined) {  
    // do something  
}
```

A.4 运算符间距

二元运算符前后必须使用一个空格来保持表达式的整洁。操作符包括赋值运算符和逻辑运算符。

```
// 好的写法  
var found = (values[i] === item);  
  
// 好的写法  
if (found && (count > 10)) {  
    doSomething();  
}  
  
// 好的写法  
for (i = 0; i < count; i++) {  
    process(i);  
}  
  
// 不好的写法：丢失了空格  
var found = (values[i]==item);  
  
// 不好的写法：丢失了空格  
if (found&&(count>10)) {  
    doSomething();  
}  
  
// 不好的写法：丢失了空格  
for (i=0; i<count; i++) {  
    process(i);  
}
```

A.5 括号间距

当使用括号时，紧接左括号之后和紧接右括号之前不应该有空格。

```
// 好的写法  
var found = (values[i] === item);  
  
// 好的写法  
if (found && (count > 10)) {  
    doSomething();  
}
```

```

// 好的写法
for (i = 0; i < count; i++) {
    process(i);
}

// 不好的写法：左括号之后有额外的空格
var found = ( values[i] === item);

// 不好的写法：右括号之前有额外的空格
if (found && (count > 10) ) {
    doSomething();
}

// 不好的写法：参数两边有额外的空格
for (i = 0; i < count; i++) {
    process( i );
}

```

A.6 对象直接量

对象直接量应当使用如下格式。

- 起始左花括号应当同表达式保持同一行。
- 每个属性的名值对应当保持一个缩进，第一个属性应当在左花括号后另起一行。
- 每个属性的名值对应当使用不含引号的属性名，其后紧跟一个冒号（之前不含空格），而后是值。
- 倘若属性值是函数类型，函数体应当在属性名之下另起一行，而且其前后均应保留一个空行。
- 一组相关的属性前后可以插入空行以提升代码的可读性。
- 结束的右花括号应当独占一行。

例如：

```

// 好的写法
var object = {

    key1: value1,

```

```

        key2: value2,
        func: function() {
            // 做些什么
        },
        key3: value3
    };

// 不好的写法：不恰当的缩进
var object = {
    key1: value1,
    key2: value2
};

// 不好的写法：函数体周围缺少空行
var object = {

    key1: value1,
    key2: value2,
    func: function() {
        // 做些什么
    },
    key3: value3
};

```

当对象字面量作为函数参数时，如果值是变量，起始花括号应当同函数名在同一行。所有其余先前列出的规则同样适用。

```

// 好的写法
doSomething({
    key1: value1,
    key2: value2
});
// 不好的写法：所有代码在一行上
doSomething({ key1: value1, key2: value2 });

```

A.7 注释

频繁地使用注释有助于他人理解你的代码。如下情况应当使用注释。

- 代码晦涩难懂。
- 可能被误认为错误的代码。

- 必要但并不明显的针对特定浏览器的代码。
- 对于对象、方法或者属性，生成文档是有必要的（使用恰当的文档注释）。

A.7.1 单行注释

单行注释应当用来说明一行代码或者一组相关的代码。单行注释可能有三种使用方式。

- 独占一行的注释，用来解释下一行代码。
- 在代码行的尾部的注释，用来解释它之前的代码。
- 多行，用来注释掉一个代码块。

这里有一些示例代码。

```
// 好的写法
if (condition) {

    // 如果代码执行到这里，则表明通过了所有安全性检查
    allowed();
}

// 不好的写法：注释之前没有空行
if (condition) {
    // 如果代码执行到这里，则表明通过了所有安全性检查
    allowed();
}

// 不好的写法：错误的缩进
if (condition) {

    // 如果代码执行到这里，则表明通过了所有安全性检查
    allowed();
}

// 不好的写法：这里应当用多行注释
// 接下来的这段代码非常难，那么，让我详细解释一下
// 这段代码的作用是首先判断条件是否为真
// 只有为真时才会执行。这里的条件是通过
// 多个函数计算出来的，在整个会话生命周期内
// 这个值是可以被修改的
if (condition) {
    // 如果代码执行到这里，则表明通过了所有安全性检查
    allowed();
}
```

对于代码行尾单行注释的情况，应确保代码结尾同注释之间至少一个缩进。

```
// 好的写法
var result = something + somethingElse; // somethingElse will never be null
```

```
// 不好的写法：代码和注释间没有足够的空格
var result = something + somethingElse; // somethingElse will never be null
```

注释一个代码块时在连续多行使用单行注释是唯一可以接受的情况。多行注释不应当在这种情况下使用。

```
// 好的写法
// if (condition){
//     doSomething();
//     thenDoSomethingElse();
// }
```

A.7.2 多行注释

多行注释应当在代码需要更多文字去解释的时候使用。每个多行注释都至少有如下三行。

1. 首行仅仅包括/*注释开始。该行不应当有其他文字。
2. 接下来的行以*开头并保持左对齐。这些行可以有文字描述。
3. 最后一行以*/开头并同先前行保持对齐。也不应当有其他文字。

多行注释的首行应当保持同它描述代码的相同层次的缩进。后续的每行应当有同样层次的缩进并附加一个空格（为了适当保持*字符的对齐）。每一个多行代码之前应当预留一个空行。

```
// 好的写法
if (condition) {

    /*
     * 如果代码执行到这里
     * 说明通过了所有的安全性检测
     */
    allowed();
}
```

```

// 不好的写法：注释之前无空行
if (condition) {
    /*
     * 如果代码执行到这里
     * 说明通过了所有的安全性检测
     */
    allowed();
}

// 不好的写法：星号后没有空格
if (condition) {
    /*
     *如果代码执行到这里
     *说明通过了所有的安全性检测
     */
    allowed();
}

// 不好的写法：错误的缩进
if (condition) {

    /*
     * 如果代码执行到这里
     * 说明通过了所有的安全性检测
     */
    allowed();
}

// 不好的写法：代码尾部注释不要用多行注释格式
var result = something + somethingElse; /*somethingElse 不应当取值为 null*/

```

A.7.3 注释声明

注释有时候也可以用来给一段代码声明额外的信息。这些声明的格式以单个单词打头并紧跟一个双引号。可使用的声明如下。

TODO

- 说明代码还未完成。应当包含下一步要做的事情。

HACK

- 表明代码实现走了一个捷径。应当包含为何使用 hack 的原因。这也可能表明

该问题可能会有更好的解决方法。

XXX

- 说明代码是有问题的并应当尽快修复。

FIXME

- 说明代码是有问题的并应尽快修复。重要性略次于 XXX。

REVIEW

- 说明代码任何可能的改动都需要评审。

这些声明可能在一行或多行注释中使用，并且应当遵循同一般注释类型相同的格式规则。

例如：

```
// 好的写法
// TODO: 我希望找到一种更快的方式
doSomething();
```

```
// 好的写法
/*
 * HACK: 不得不针对 IE 做的特殊处理. 我计划后续有时间时
 * 重写这部分. 这些代码可能需要在 v1.2 版本之前替换掉.
 */
if (document.all) {
    doSomething();
}
```

```
// 好的写法
// REVIEW: 有更好的方法吗?
if (document.all){
    doSomething();
}
```

```
// 不好的写法: 注释声明空格不正确
// TODO : 我希望找到一种更快的方式
doSomething();
```

```
// 不好的写法: 代码和注释应当保持同样的缩进
// REVIEW: 有更好的方法吗?
if (document.all) {
    doSomething();
}
```

A.7.4 变量声明

所有的变量在使用前都应当事先定义。变量定义应当放在函数开头，使用一个 var 表达式^①每行一个变量。除了首行，所有行都应当多一层缩进以使变量名能够垂直方向对齐。变量定义时应当初始化，并且赋值操作符应当保持一致的缩进。初始化的变量应当在未初始化变量之前。

```
// 好的写法
var count = 10,
    name = "Nicholas",
    found = false,
    empty;

// 不好的写法：不恰当的初始化赋值
var count = 10,
    name = "Nicholas",
    found= false,
    empty;

// 不好的写法：错误的缩进
var count = 10,
name = "Nicholas",
found = false,
empty;

// 不好的写法：多个定义写在一行
var count = 10, name = "Nicholas",
    found = false, empty;

// 不好的写法：未初始化的变量放在最前边
var empty,
    count = 10,
    name = "Nicholas",
    found = false;

// 不好的写法：多个 var 表达式
var count = 10,
    name = "Nicholas";

var found = false,
    empty;
```

^① 译注：在《JavaScript Patterns》（O'Reilly 出版 2010 年出版）书中也推荐使用单 var 语句，并将其定义为一种最佳实践，《JS Pattern》中作者细致充分地讲解了声明提前和单 var 模式。

A.7.5 函数声明

函数应当在使用前声明。一个不是作为方法的函数（也就是说没有作为一个对象的属性）应当使用函数声明的格式（不是函数表达式和 Function 构造器格式）。函数名和开始圆括号之间不应当有空格。结束的圆括号和右边的花括号之间应该留一个空格。右侧的花括号应当同 function 关键字保持同一行。开始和结束括号之间不应该有空格。参数名之间应当在逗号之后保留一个空格。函数体应当保持一级缩进。

```
// 好的写法
function doSomething(arg1, arg2) {
    return arg1 + arg2;
}

// 不好的写法：第一行不恰当的空格
function doSomething (arg1, arg2){
    return arg1 + arg2;
}

// 不好的写法：函数表达式
var doSomething = function(arg1, arg2) {
    return arg1 + arg2;
};

// 不好的写法：左侧的花括号位置不对
function doSomething(arg1, arg2)
{
    return arg1 + arg2;
}
// 错误的写法：使用了 Function 构造器
var doSomething = new Function("arg1", "arg2", "return arg1 + arg2");
```

其他函数内部定义的函数应当在 var 语句后立即定义。

```
// 好的写法
function outer() {

    var count = 10,
        name = "Nicholas",
        found = false,
        empty;

    function inner() {
        // 代码
    }
}
```

```
// 调用 inner() 的代码
}

// 不好的写法：inner 函数的定义先于变量
function outer() {

    function inner() {
        // 代码
    }

    var count = 10,
        name = "Nicholas",
        found = false,
        empty;

    // 调用 inner() 的代码
}
```

匿名函数可能作为方法赋值给对象，或者作为其他函数的参数。function 关键字同开始括号之间不应有空格。

```
// 好的写法
object.method = function() {
    // 代码
};

// 不好的写法：不正确的空格
object.method = function () {
    // 代码
};
```

立即被调用的函数应当在函数调用的外层用圆括号包裹。

```
// 好的写法
var value = (function() {

    // 函数体

    return {
        message: "Hi"
    }
})();

// 不好的写法：函数调用外层没有用圆括号包裹
var value = function() {

    // function body

    return {
        message: "Hi"
    }
}
```

```
}());
// 不好的写法：圆括号位置不当
var value = (function() {
    // 函数体

    return {
        message: "Hi"
    }
})();
```

A.7.6 命名

变量和函数在命名时应当小心。命名应仅限于数字字母字符，某些情况下也可以使用下划线。最好不要在任何命名中使用美元符号(\$)或者反斜杠(\)。

变量命名应当采用驼峰命名格式，首字母小写，每个单词首字母大写。变量名的第一个单词应当是一个名词（而非动词）以避免同函数混淆。不要在变量命名中使用下划线。

```
// 好的写法
var accountNumber = "8401-1";

// 不好的写法：大写字母开头
var AccountNumber = "8401-1";

// 不好的写法：动词开头
var getAccountNumber = "8401-1";

// 不好的写法：使用下划线
var account_number = "8401-1";
```

函数命名也应当采用驼峰命名格式。函数名的第一个单词应当是动词（而非名词）来避免同变量混淆。函数名中最好不要使用下划线。

```
// 好的写法
function doSomething() {
    // 代码
}

// 不好的写法：大写字母开头
function DoSomething() {
    // 代码
}
```

```
// 不好的写法：名词开头
function car() {
    // code
}

// 不好的写法：使用下划线
function do_something() {
    // code
}
```

构造函数——通过 new 运算符创建新对象的函数——也应当以驼峰格式^①命名并且首字符大写。构造函数名称应当以非动词开头，因为 new 代表着创建一个对象实例的操作。

```
// 好的写法
function MyObject() {
    // 代码
}

// 不好的写法：小写字母开头
function myObject() {
    // code
}

// 不好的写法：使用下划线
function My_Object() {
    // code
}

// 不好的写法：动词开头
function getMyObject() {
    // code
}
```

常量（值不会被改变的变量）的命名应当是所有字母大写，不同单词之间用单个下划线隔开。

```
// 好的写法
var TOTAL_COUNT = 10;

// 不好的写法：驼峰形式
var totalCount = 10;

// 不好的写法：混合形式
```

^① 译注：驼峰式大小写（Camel-Case）一词来自 Perl 语言中普遍使用的大小写混合格式，分为小驼峰和大驼峰两种命名形式。

```
var total_COUNT = 10;
```

对象的属性同变量的命名规则相同。对象的方法同函数的命名规则相同。如果属性或者方法是私有的，应当在之前加一个下划线。

```
// 好的写法
var object = {
    _count: 10,
    _getCount: function () {
        return this._count;
    }
};
```

A.7.7 严格模式

严格模式应当仅限在函数内部使用，千万不要在全局使用。

```
// 不好的写法：全局使用严格模式
"use strict";

function doSomething() {
    // 代码
}

// 好的写法
function doSomething() {
    "use strict";
    // 代码
}
```

如果你期望在多个函数中使用严格模式而不需要多次声明“use strict”，可以使用立即被调用的函数。

```
// 好的写法
(function() {
    "use strict";

    function doSomething() {
        // 代码
    }

    function doSomethingElse() {
        // 代码
    }
});
```

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

A.7.8 赋值

当给变量赋值时，如果右侧是含有比较语句的表达式，需要用圆括号包裹。

```
// 好的写法  
var flag = (i < count);  
  
// 不好的写法：遗漏圆括号  
var flag = i < count;
```

A.7.9 等号运算符

使用`==`（严格相等）和`!=`（严格不相等）代替`=`（相等）和`!=`（不等）来避免弱类型转换错误。

```
// 好的写法  
var same = (a === b);  
  
// 不好的写法：使用 ==  
var same = (a == b);
```

A.7.10 三元操作符

三元运算符应当仅仅用在条件赋值语句中，而不要作为`if`语句的替代品。

```
// 好的写法  
var value = condition ? value1 : value2;  
  
// 不好的写法：没有赋值，应当使用 if 表达式  
condition ? doSomething() : doSomethingElse();
```

A.7.11 语句

简单语句

每一行最多只包含一条语句。所有简单的语句都应该以分号`(;)`结束。

```
// 好的写法  
count++;  
a = b;
```

```
// 不好的写法：多个表达式写在一行  
count++; a = b;
```

返回语句

返回语句当返回一个值的时候不应当使用圆括号包裹，除非在某些情况下这么做可以让返回值更容易理解。例如：

```
return;  
  
return collection.size();  
  
return (size > 0 ? size : defaultSize);
```

复合语句

复合语句是大括号括起来的语句列表。

- 括起来的语句应当较复合语句多缩进一个层级。
- 开始的大括号应当在复合语句所在行的末尾；结束的大括号应当独占一行且同复合语句的开始保持同样的缩进。
- 当语句是控制结构的一部分时，诸如 if 或者 for 语句，所有语句都需要用大括号括起来，也包括单个语句。这个约定使得我们更方便地添加语句而不用担心忘记加括号而引起 bug。
- 像 if 一样的语句开始的关键词，其后应该紧跟一个空格，起始大括号应当在空格之后。

if 语句

if 语句应当是下面的格式。

```
if (condition) {  
    statements  
}  
  
if (condition) {  
    statements  
} else {  
    statements  
}
```

```
if (condition) {  
    statements  
} else if (condition) {  
    statements  
} else {  
    statements  
}
```

绝不允许在 if 语句中省略花括号。

```
// 好的写法  
if (condition) {  
    doSomething();  
}  
  
// 不好的写法：不恰当的空格  
if(condition){  
    doSomething();  
}  
  
// 不好的写法：遗漏花括号  
if (condition)  
    doSomething();  
  
// 不好的写法：所有代码写在一行  
if (condition) { doSomething(); }  
  
// 不好的写法：所有代码写在一行且没有花括号  
if (condition) doSomething();
```

for 语句

for 类型的语句应当是下面的格式。

```
for (initialization; condition; update) {  
    statements  
}  
  
for (variable in object) {  
    statements  
}
```

for 语句的初始化部分不应当有变量声明。

```
// 好的写法  
var i,  
    len;
```

```
for (i=0, len=10; i < len; i++) {  
    // 代码  
}  
  
// 不好的写法：初始化时候声明变量  
for (var i=0, len=10; i < len; i++) {  
    // code  
}  
  
// 不好的写法：初始化时候声明变量  
for (var prop in object) {  
    // code  
}
```

当使用 `for-in` 语句时，记得使用 `hasOwnProperty()` 进行双重检查来过滤出对象的成员。

while 语句

`while` 类的语句应当是下面的格式。

```
while (condition) {  
    statements  
}
```

do 语句

`do` 类的语句应当是下面的格式。

```
do {  
    statements  
} while (condition);
```

switch 语句

`switch` 类的语句应当是如下格式。

```
switch (expression) {  
    case expression:  
        statements  
  
    default:  
        statements  
}
```

`switch` 下的每一个 `case` 都应当保持一个缩进。除第一个之外包括 `default` 在内的每一个 `case` 都应当在之前保持一个空行。

每一组语句（除了 default）都应当以 break、return、throw 结尾，或者用一行注释表示跳过。

```
// 好的写法
switch (value) {
    case 1:
        /* falls through */

    case 2:
        doSomething();
        break;

    case 3:
        return true;

    default:
        throw new Error("This shouldn't happen.");
}
```

如果一个 switch 语句不包含 default 情况，应当用一行注释代替。

```
// 好的写法
switch (value) {
    case 1:
        /*falls through*/

    case 2:
        doSomething();
        break;

    case 3:
        return true;

    // 没有 default
}
```

try 语句

try 类的语句应当格式如下。

```
try {
    statements
} catch (variable) {
    statements
}
```

```
try {
    statements
} catch (variable) {
    statements
} finally {
    statements
}
```

A.7.12 留白

在逻辑相关的代码块之间添加空行可以提高代码的可读性。

两行空行仅限在如下情况中使用。

- 在不同的源代码文件之间。
- 在类和接口定义之间。

单行空行仅限在如下情况中使用。

- 方法之间。
- 方法中局部变量和第一行语句之间。
- 多行或者单行注释之前。
- 方法中逻辑代码块之间以提升代码的可读性。

空格应当在如下情况中使用。

- 关键词后跟括号的情况应当用空格隔开。
- 参数列表中逗号之后应当保留一个空格。
- 所有的除了点(.)之外的二元运算符，其操作数都应当用空格隔开。单目运算符的操作数之间不应该用空白隔开，诸如一元减号，递增(++)，递减(--)。
- for语句中的表达式之间应当用空格隔开。

A.7.13 需要避免的

- 切勿使用像 String 一类的原始包装类型创建新的对象。
- 避免使用 eval()。
- 避免使用 with 语句。该语句在严格模式中不复存在，可能在未来的 ECMAScript 标准中也将去除。

JavaScript 工具集

B.1 构建工具

尽管很多构建工具并非只针对 JavaScript 的，但是它们对于管理你的大型 JavaScript 项目同样适用。

Ant (<http://ant.apache.org>)

我最喜欢的 JavaScript 构建工具。一个基于 Java 的构建系统。

Buildy (<https://github.com/.mosen/buildy>)

一个原生支持 JavaScript 和 CSS 相关任务的基于 Node.js 的构建系统。

Gmake (<http://www.gnu.org/s/make/>)

一个在 Unix 贡献者之间依旧很受欢迎的较老的构建工具。Gmake 被 jQuery 所使用。

Grunt (<https://github.com/cowboy/grunt>)

一个基于 Node.js 的构建系统，原生支持 JavaScript 相关的任务，诸如压缩和合并。

Jammit (<http://documentcloud.github.com/jammit/>)

一个基于 Ruby 的处理压缩，校验等的静态包。

Jasy (<https://github.com/zynga/jasy>)

一个基于 Python 的构建系统。

Rake (<http://rake.rubyforge.org/>)

一个用 Ruby 写的类似 Gmake 的实用工具。项目使用了 Sass，一个很受欢迎的 CSS 预解释器，倾向于使用 Rake。

Sprockets (<http://getsprockets.org>)

一个基于 Rack 的构建系统。

B.2 文档生成器

文档生成器用来根据源代码中的注释生成文档。

Docco (<http://jashkenas.github.com/cocco/>)

一个一对一的文档生成器，根据代码显示文档。用 CoffeeScript 编写。

Dojo Documentation Tools (<http://dojotoolkit.org/reference-guide/util/doctools.html>)

Dojo 的官方文档生成器。采用 PHP 编写。

JoDoc (<https://github.com/azakus/joodoc-js>)

一个使用 Markdown 语法的 JavaScript 文档生成器。采用 JavaScript 编写。

JS Doc ToolKit (<http://code.google.com/p/jsdoc-toolkit/>)

基于 Java 的文档生成器。使用最频繁的文档生成器之一。

Natural Docs (<http://www.naturaldocs.org>)

以通用为目标的兼容多种语言的文档生成器。采用 Perl 语言编写。

NDoc (<https://github.com/modeca/ndoc>)

PDoc 的一个 JavaScript 接口。

PDoc (<http://pdoc.org/>)

Prototype 的官方文档生成器。采用 Ruby 语言编写。

YUI Doc (<http://yuilibrary.com/projects/yuidoc>)

YUI 文档生成工具。采用 JavaScript 语言编写。

B.3 代码检查工具

代码检查工具有助于识别代码中有问题的语句和风格。

JSLint (<http://jslint.com>)

Crockford 的代码质量检查工具。

JSHint (<http://jslint.com>)

JSLint 的分支版本，添加了更多的配置选项。

B.4 压缩工具

压缩工具可以去除不必要的注释和空格，也可以执行其他的代码优化，从而使 JavaScript 文件体积更小。

Closure Compiler (<http://code.google.com/closure/compiler/>)

Google 的一个基于 Java 的压缩工具。

UglifyJS (<https://github.com/mishoo/UglifyJS>)

一个基于 Node.js 的 JavaScript 压缩工具。

YUI Compressor (<http://yuilibrary.com/projects/yuicompressor>)

一个基于 Java 的 JavaScript 和 CSS 压缩工具。

B.5 测试工具

通过测试工具可以使你方便地书写和执行测试用例来验证代码的行为。

Jasmine (<http://pivotal.github.com/jasmine/>)

一个行为驱动的测试框架。

JsTestDriver (<http://code.google.com/p/js-test-driver/>)

Google 的单元测试框架，内置浏览器自动化测试。

PhantomJS (<http://www.phantomjs.org>)

一个为测试而生的命令行版本的 Webkit 浏览器。通过驱动系统可以兼容 QUnit 和 Jasmine（默认）和其他的测试框架。

QUnit (<http://docs.jquery.com/QUnit>)

jQuery 的测试框架。

Selenium (<http://seleniumhq.com>)

可以用来执行浏览器测试的功能性测试框架。

Yeti (<http://yuilibrary.com/projects/yeti>)

浏览器中测试 JavaScript 的测试工具。

YUI Test (<http://yuilibrary.com/projects/yuitest>)

YUI 单元测试框架。

封面介绍

这本书的封面动物是希腊陆龟（Greek tortoise）。

希腊陆龟（欧洲陆龟）也被称为股刺陆龟，它有至少 20 种已知的亚物种。它体型大，重量大，颜色深。它们的栖息地为北非、南欧和东南亚，这些地方比较炎热干旱，但也偶有生活于山区的干草地和海岸沙丘之中。它们基因的多样性导致它们很难被归类，而且通常会发生异种交配。不同种类的乌龟之间进行交配，会导致它们的后代在体型和颜色上差别甚大。正因为此，识别乌龟种类的最好方法就是搞清楚它们是来自哪里。

希腊陆龟的体型尺寸从 8 英寸到 12 英寸不等。它们大腿上的“鳞刺”和两种小结核有关系，这种小结核生长于尾部的两侧，通常也会在背部甲壳的长方形的“块”上见到。通常我们会从它们粗壮的前腿、斑点脊椎和肋板等特征来识别它们，它们的特点还包括巨大的背部斑点，它的甲壳直到尾部都是不可分的。

希腊陆龟在冬眠醒来之后会立即开始本能的交配。产卵之前会有一到两周的时间迁移它们的栖息地，它们通过挖、尝、闻泥土来寻找理想的产卵地点。在产卵之前一两天的时间里，雌性希腊陆龟会变得很有攻击性，以保证它的领地不会被袭扰，从而保证卵的安全。它们的平均寿命在 50 年左右。

