

## 第9章

# 客户端检测

### 本章内容

- 使用能力检测
- 用户代理检测的历史
- 选择检测方式

浏览器提供商虽然在实现公共接口方面投入了很多精力，但结果仍然是每一种浏览器都有各自的长处，也都有各自的缺点。即使是那些跨平台的浏览器，虽然从技术上看版本相同，也照样存在不一致性问题。面对普遍存在的不一致性问题，开发人员要么采取迁就各方的“最小公分母”策略，要么（也是更常见的）就得利用各种客户端检测方法，来突破或者规避种种局限性。

迄今为止，客户端检测仍然是 Web 开发领域中一个饱受争议的话题。一谈到这个话题，人们总会不约而同地提到浏览器应该支持一组最常用的公共功能。在理想状态下，确实应该如此。但是，在现实当中，浏览器之间的差异以及不同浏览器的“怪癖”（quirk），多得简直不胜枚举。因此，客户端检测除了是一种补救措施之外，更是一种行之有效的开发策略。

检测 Web 客户端的手段很多，而且各有利弊。但最重要的还是要知道，不到万不得已，就不要使用客户端检测。只要能找到更通用的方法，就应该优先采用更通用的方法。一言以蔽之，先设计最通用的方案，然后再使用特定于浏览器的技术增强该方案。

## 9.1 能力检测

最常用也最为人们广泛接受的客户端检测形式是能力检测（又称特性检测）。能力检测的目标不是识别特定的浏览器，而是识别浏览器的能力。采用这种方式不必顾及特定的浏览器如何如何，只要确定浏览器支持特定的能力，就可以给出解决方案。能力检测的基本模式如下：

```
if (object.propertyInQuestion){
    //使用 object.propertyInQuestion
}
```

举例来说，IE5.0 之前的版本不支持 `document.getElementById()` 这个 DOM 方法。尽管可以使用非标准的 `document.all` 属性实现相同的目的，但 IE 的早期版本中确实不存在 `document.getElementById()`。于是，也就有了类似下面的能力检测代码：

```
function getElement(id){
    if (document.getElementById){
        return document.getElementById(id);
    } else if (document.all){
        return document.all[id];
    }
}
```

```

    } else {
        throw new Error("No way to retrieve element!");
    }
}

```

这里的 `getElement()` 函数的用途是返回具有给定 ID 的元素。因为 `document.getElementById()` 是实现这一目的的标准方式，所以一开始就测试了这个方法。如果该函数存在（不是未定义），则使用该函数。否则，就要继续检测 `document.all` 是否存在，如果是，则使用它。如果上述两个特性都不存在（很有可能），则创建并抛出错误，表示这个函数无法使用。

要理解能力检测，首先必须理解两个重要的概念。如前所述，第一个概念就是先检测达成目的的最常用的特性。对前面的例子来说，就是要先检测 `document.getElementById()`，后检测 `document.all`。先检测最常用的特性可以保证代码最优化，因为在多数情况下都可以避免测试多个条件。

第二个重要的概念就是必须测试实际要用到的特性。一个特性存在，不一定意味着另一个特性也存在。来看一个例子：

```

function getWindowWidth(){
    if (document.all){ //假设是 IE
        return document.documentElement.clientWidth; //错误的用法!!!
    } else {
        return window.innerWidth;
    }
}

```

这是一个错误使用能力检测的例子。`getWindowWidth()` 函数首先检查 `document.all` 是否存在，如果是则返回 `document.documentElement.clientWidth`。第 8 章曾经讨论过，IE8 及之前版本确实不支持 `window.innerWidth` 属性。但问题是 `document.all` 存在也不一定表示浏览器就是 IE。实际上，也可能是 Opera；Opera 支持 `document.all`，也支持 `window.innerWidth`。

## 9.1.1 更可靠的能力检测

能力检测对于想知道某个特性是否会按照适当方式行事（而不仅仅是某个特性存在）非常有用。上一节中的例子利用类型转换来确定某个对象成员是否存在，但这样你还是不知道该成员是不是你想要的。来看下面的函数，它用来确定一个对象是否支持排序。

```

//不要这样做！这不是能力检测——只检测了是否存在相应的方法
function isSortable(object){
    return !!object.sort;
}

```

这个函数通过检测对象是否存在 `sort()` 方法，来确定对象是否支持排序。问题是，任何包含 `sort` 属性的对象也会返回 `true`。

```
var result = isSortable({ sort: true });
```

检测某个属性是否存在并不能确定对象是否支持排序。更好的方式是检测 `sort` 是不是一个函数。

```

//这样更好：检查 sort 是不是函数
function isSortable(object){
    return typeof object.sort == "function";
}

```

这里的 `typeof` 操作符用于确定 `sort` 的确是一个函数，因此可以调用它对数据进行排序。

在可能的情况下，要尽量使用 `typeof` 进行能力检测。特别是，宿主对象没有义务让 `typeof` 返回合理的值。最令人发指的事儿就发生在 IE 中。大多数浏览器在检测到 `document.createElement()` 存在时，都会返回 `true`。

```
//在 IE8 及之前版本中不行
function hasCreateElement(){
    return typeof document.createElement == "function";
}
```

在 IE8 及之前版本中，这个函数返回 `false`，因为 `typeof document.createElement` 返回的是 `"object"`，而不是 `"function"`。如前所述，DOM 对象是宿主对象，IE 及更早版本中的宿主对象是通过 COM 而非 JavaScript 实现的。因此，`document.createElement()` 函数确实是一个 COM 对象，所以 `typeof` 才会返回 `"object"`。IE9 纠正了这个问题，对所有 DOM 方法都返回 `"function"`。

关于 `typeof` 的行为不标准，IE 中还可以举出例子来。ActiveX 对象（只有 IE 支持）与其他对象的行为差异很大。例如，不使用 `typeof` 测试某个属性会导致错误，如下所示。

```
//在 IE 中会导致错误
var xhr = new XMLHttpRequest("Microsoft.XMLHttp");
if (xhr.open){ //这里会发生错误
    //执行操作
}
```

像这样直接把函数作为属性访问会导致 JavaScript 错误。使用 `typeof` 操作符会更靠谱一点，但 IE 对 `typeof xhr.open` 会返回 `"unknown"`。这就意味着，在浏览器环境下测试任何对象的某个特性是否存在，要使用下面这个函数。

```
//作者: Peter Michaux
function isHostMethod(object, property) {
    var t = typeof object[property];
    return t=='function' ||
        (!!(!t=='object' && object[property])) ||
        t=='unknown';
}
```

可以像下面这样使用这个函数：

```
result = isHostMethod(xhr, "open"); //true
result = isHostMethod(xhr, "foo"); //false
```

目前使用 `isHostMethod()` 方法还是比较可靠的，因为它考虑到了浏览器的怪异行为。不过也要注意，宿主对象没有义务保持目前的实现方式不变，也不一定会模仿已有宿主对象的行为。所以，这个函数——以及其他类似函数，都不能百分之百地保证永远可靠。作为开发人员，必须对自己要使用某个功能的风险作出理性的估计。



要想深入了解围绕 JavaScript 中能力检测的一些观点，请参考 Peter Michaux 的文章“Feature Detection: State of the Art Browser Scripting”，网址为 <http://peter.michaux.ca/articles/feature-detection-state-of-the-art-browser-scripting>。

## 9.1.2 能力检测，不是浏览器检测

检测某个或某几个特性并不能够确定浏览器。下面给出的这段代码（或与之差不多的代码）可以在许多网站中看到，这种“浏览器检测”代码就是错误地依赖能力检测的典型示例。

```
//错误！还不够具体
var isFirefox = !(navigator.vendor && navigator.vendorSub);

//错误！假设过头了
var isIE = !(document.all && document.uniqueID);
```

这两行代码代表了对能力检测的典型误用。以前，确实可以通过检测 `navigator.vendor` 和 `navigator.vendorSub` 来确定 Firefox 浏览器。但是，Safari 也依葫芦画瓢地实现了相同的属性。于是，这段代码就会导致人们作出错误的判断。为检测 IE，代码测试了 `document.all` 和 `document.uniqueID`。这就相当于假设 IE 将来的版本中仍然会继续存在这两个属性，同时还假设其他浏览器都不会实现这两个属性。最后，这两个检测都使用了双逻辑非操作符来得到布尔值（比先存储后访问的效果更好）。

实际上，根据浏览器不同将能力组合起来是更可取的方式。如果你知道你的应用程序需要使用某些特定的浏览器特性，那么最好是一次性检测所有相关特性，而不要分别检测。看下面的例子。

```
//确定浏览器是否支持 Netscape 风格的插件
var hasNSPlugins = !(navigator.plugins && navigator.plugins.length);

//确定浏览器是否具有 DOM1 级规定的的能力
var hasDOM1 = !(document.getElementById && document.createElement &&
    document.getElementsByTagName);
```

*CapabilitiesDetectionExample01.htm*

以上例子展示了两个检测：一个检测浏览器是否支持 Netscape 风格的插件；另一个检测浏览器是否具备 DOM1 级所规定的的能力。得到的布尔值可以在以后继续使用，从而节省重新检测能力的时间。



在实际开发中，应该将能力检测作为确定下一步解决方案的依据，而不是用它来判断用户使用的是哪种浏览器。

## 9.2 怪癖检测

与能力检测类似，怪癖检测（quirks detection）的目标是识别浏览器的特殊行为。但与能力检测确认浏览器支持什么能力不同，怪癖检测是想要知道浏览器存在什么缺陷（“怪癖”也就是 bug）。这通常需要运行一小段代码，以确定某一特性不能正常工作。例如，IE8 及更早版本中存在一个 bug，即如果某个实例属性与标记为 `[DontEnum]` 的某个原型属性同名，那么该实例属性将不会出现在 `for-in` 循环当中。可以使用如下代码来检测这种“怪癖”。

```
var hasDontEnumQuirk = function(){

    var o = { toString : function(){} };
    for (var prop in o){
```

```

        if (prop == "toString"){
            return false;
        }

        return true;
    }();

```

QuirksDetectionExample01.htm

以上代码通过一个匿名函数来测试该“怪癖”，函数中创建了一个带有 `toString()` 方法的对象。在正确的 ECMAScript 实现中，`toString` 应该在 `for-in` 循环中作为属性返回。

另一个经常需要检测的“怪癖”是 Safari 3 以前版本会枚举被隐藏的属性。可以用下面的函数来检测该“怪癖”。

```

var hasEnumShadowsQuirk = function(){

    var o = { toString : function(){} };
    var count = 0;
    for (var prop in o){
        if (prop == "toString"){
            count++;
        }
    }

    return (count > 1);
}();

```

QuirksDetectionExample01.htm

如果浏览器存在这个 bug，那么使用 `for-in` 循环枚举带有自定义的 `toString()` 方法的对象，就会返回两个 `toString` 的实例。

一般来说，“怪癖”都是个别浏览器所独有的，而且通常被归为 bug。在相关浏览器的新版本中，这些问题可能会也可能不会被修复。由于检测“怪癖”涉及运行代码，因此我们建议仅检测那些对你有直接影响的“怪癖”，而且最好在脚本一开始就执行此类检测，以便尽早解决问题。

## 9.3 用户代理检测

第三种，也是争议最大的一种客户端检测技术叫做用户代理检测。用户代理检测通过检测用户代理字符串来确定实际使用的浏览器。在每一次 HTTP 请求过程中，用户代理字符串是作为响应首部发送的，而且该字符串可以通过 JavaScript 的 `navigator.userAgent` 属性访问。在服务器端，通过检测用户代理字符串来确定用户使用的浏览器是一种常用而且广为接受的做法。而在客户端，用户代理检测一般被当作一种万不得已才用的做法，其优先级排在能力检测和（或）怪癖检测之后。

提到与用户代理字符串有关的争议，就不得不提到电子欺骗（spoofing）。所谓电子欺骗，就是指浏览器通过在自己的用户代理字符串加入一些错误或误导性信息，来达到欺骗服务器的目的。要弄清楚这个问题的来龙去脉，必须从 Web 问世初期用户代理字符串的发展讲起。

## 9.3.1 用户代理字符串的历史

HTTP 规范（包括 1.0 和 1.1 版）明确规定，浏览器应该发送简短的用户代理字符串，指明浏览器的名称和版本号。RFC 2616（即 HTTP 1.1 协议规范）是这样描述用户代理字符串的：

“产品标识符常用于通信应用程序标识自身，由软件名和版本组成。使用产品标识符的大多数领域也允许列出作为应用程序主要部分的子产品，由空格分隔。按照惯例，产品要按照相应的重要程度依次列出，以便标识应用程序。”

上述规范进一步规定，用户代理字符串应该以一组产品的形式给出，字符串格式为：标识符/产品版本号。但是，现实中的用户代理字符串则绝没有如此简单。

### 1. 早期的浏览器

1993 年，美国 NCSA（National Center for Supercomputing Applications，国家超级计算机中心）发布了世界上第一款 Web 浏览器 Mosaic。这款浏览器的用户代理字符串非常简单，类似如下所示。

```
Mosaic/0.9
```

尽管这个字符串在不同操作系统和不同平台下会有所变化，但其基本格式还是简单明了的。正斜杠前面的文本表示产品名称（有时候会出现 NCSA Mosaic 或其他类似字样），而斜杠后面的文本是产品的版本号。

Netscape Communications 公司介入浏览器开发领域后，遂将自己产品的代号定名为 Mozilla（Mosaic Killer 的简写，意即 Mosaic 杀手）。该公司第一个公开发版，Netscape Navigator 2 的用户代理字符串具有如下格式。

```
Mozilla/版本号 [语言] (平台; 加密类型)
```

Netscape 在坚持将产品名和版本号作为用户代理字符串开头的基础上，又在后面依次添加了下列信息。

- 语言：即语言代码，表示应用程序针对哪种语言设计。
- 平台：即操作系统和（或）平台，表示应用程序的运行环境。
- 加密类型：即安全加密的类型。可能的值有 U（128 位加密）、I（40 位加密）和 N（未加密）。典型的 Netscape Navigator 2 的用户代理字符串如下所示。

```
Mozilla/2.02 [fr] (WinNT; I)
```

这个字符串表示浏览器是 Netscape Navigator 2.02，为法语国家编译，运行在 Windows NT 平台下，加密类型为 40 位。那个时候，通过用户代理字符串中的产品名称，至少还能够轻易地确定用户使用的是哪种浏览器。

### 2. Netscape Navigator 3 和 Internet Explorer 3

1996 年，Netscape Navigator 3 发布，随即超越 Mosaic 成为当时最流行的 Web 浏览器。而用户代理字符串只作了一些小的改变，删除了语言标记，同时允许添加操作系统或系统使用的 CPU 等可选信息。于是，格式变成如下所示。

```
Mozilla/版本号 (平台; 加密类型 [; 操作系统或 CPU 说明])
```

运行在 Windows 系统下的 Netscape Navigator 3 的用户代理字符串大致如下。

```
Mozilla/3.0 (Win95; U)
```

这个字符串表示 Netscape Navigator 3 运行在 Windows 95 中，采用了 128 位加密技术。可见，在 Windows 系统中，字符串中的操作系统或 CPU 说明被省略了。

Netscape Navigator 3 发布后不久，微软也发布了其第一款赢得用户广泛认可的 Web 浏览器，即 Internet Explorer 3。由于 Netscape 浏览器在当时占绝对市场份额，许多服务器在提供网页之前都要专门检测该浏览器。如果用户通过 IE 打不开相关网页，那么这个新生的浏览器很可能就会夭折。于是，微软决定将 IE 的用户代理字符串修改成兼容 Netscape 的形式，结果如下：

```
Mozilla/2.0 (compatible; MSIE 版本号; 操作系统)
```

例如，Windows 95 平台下的 Internet Explorer 3.02 带有如下用户代理字符串：

```
Mozilla/2.0 (compatible; MSIE 3.02; Windows 95)
```

由于当时的大多数浏览器嗅探程序只检测用户代理字符串中的产品名称部分，结果 IE 就成功地将自己标识为 Mozilla，从而伪装成 Netscape Navigator。微软的这一做法招致了很多批评，因为它违反了浏览器标识的惯例。更不规范的是，IE 将真正的浏览器版本号插入到了字符串的中间。

字符串中另外一个有趣的地方是标识符 Mozilla 2.0（而不是 3.0）。毕竟，当时的主流版本是 3.0，改成 3.0 应该对微软更有利才对。但真正的谜底到现在还没有揭开——但很可能只是人为疏忽所致。

### 3. Netscape Communicator 4 和 IE4~IE8

1997 年 8 月，Netscape Communicator 4 发布（这一版将浏览器名字中的 Navigator 换成了 Communicator）。Netscape 继续遵循了第 3 版时的用户代理字符串格式：

```
Mozilla/版本号 (平台; 加密类型 [; 操作系统或 CPU 说明])
```

因此，Windows 98 平台中第 4 版的用户代理字符串如下所示：

```
Mozilla/4.0 (Win98; I)
```

Netscape 在发布补丁时，子版本号也会相应提高，用户代理字符串如下面的 4.79 版所示：

```
Mozilla/4.79 (Win98; I)
```

但是，微软在发布 Internet Explorer 4 时，顺便将用户代理字符串修改成了如下格式：

```
Mozilla/4.0 (compatible; MSIE 版本号; 操作系统)
```

换句话说，对于 Windows 98 中运行的 IE4 而言，其用户代理字符串为：

```
Mozilla/4.0 (compatible; MSIE 4.0; Windows 98)
```

经过此番修改，Mozilla 版本号就与实际的 IE 版本号一致了，为识别它们的第四代浏览器提供了方便。但令人遗憾的是，两者的一致性仅限于这一个版本。在 Internet Explorer 4.5 发布时（只针对 Macs），虽然 Mozilla 版本号还是 4，但 IE 版本号则改成了如下所示：

```
Mozilla/4.0 (compatible; MSIE 4.5; Mac_PowerPC)
```

此后，IE 的版本一直到 7 都沿袭了这个模式：

```
Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)
```

而 IE8 的用户代理字符串中添加了呈现引擎（Trident）的版本号：

```
Mozilla/4.0 (compatible; MSIE 版本号; 操作系统; Trident/Trident 版本号)
```

例如:

```
Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0)
```

这个新增的 **Trident** 记号是为了让开发人员知道 IE8 是不是在兼容模式下运行。如果是, 则 **MSIE** 的版本号会变成 7, 但 **Trident** 及版本号还会留在用户代码字符串中:

```
Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/4.0)
```

增加这个记号有助于分辨浏览器到底是 IE7 (没有 **Trident** 记号), 还是运行在兼容模式下的 IE8。

IE9 对字符串格式做了一点调整。**Mozilla** 版本号增加到了 5.0, 而 **Trident** 的版本号也升到了 5.0。IE9 默认的用户代理字符串如下:

```
Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0)
```

如果 IE9 运行在兼容模式下, 字符串中的 **Mozilla** 版本号和 **MSIE** 版本号会恢复旧的值, 但 **Trident** 的版本号仍然是 5.0。例如, 下面就是 IE9 运行在 IE7 兼容模式下的用户代理字符串:

```
Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.1; Trident/5.0)
```

所有这些变化都是为了确保过去的用户代理检测脚本能够继续发挥作用, 同时还能给新脚本提供更丰富的信息。

#### 4. Gecko

Gecko 是 Firefox 的呈现引擎。当初的 Gecko 是作为通用 Mozilla 浏览器的一部分开发的, 而第一个采用 Gecko 引擎的浏览器是 Netscape 6。为 Netscape 6 编写的一份规范中规定了未来版本中用户代理字符串的构成。这个新格式与 4.x 版本中相对简单的字符串相比, 有着非常大的区别, 如下所示:

```
Mozilla/Mozilla 版本号 (平台; 加密类型; 操作系统或 CPU; 语言; 预先发行版本)
Gecko/Gecko 版本号 应用程序或产品/应用程序或产品版本号
```

这个明显复杂了很多的用户代理字符串中蕴含很多新想法。下表列出了字符串中各项的用意。

字符串项	必需吗	说 明
Mozilla版本号	是	Mozilla的版本号
平台	是	浏览器运行的平台。可能的值包括Windows、Mac和X11 (指Unix的X窗口系统)
加密类型	是	加密技术的类型: U表示128位、I表示40位、N表示未加密
操作系统或CPU	是	浏览器运行的操作系统或计算机系统使用的CPU。在Windows平台中, 这一项指Windows的版本 (如WinNT、Win95, 等等)。如果平台是Macintosh, 这一项指CPU (针对PowerPC的68K、PPC, 或MacIntel)。如果平台是X11, 这一项是Unix操作系统的名称, 与使用Unix命令uname-sm得到的名称相同
语言	是	浏览器设计时所针对的目标用户语言
预先发行版本	否	最初用于表示Mozilla的预先发行版本, 现在则用来表示Gecko呈现引擎的版本号
Gecko版本号	是	Gecko呈现引擎的版本号, 但由yyyymmdd格式的日期表示
应用程序或产品	否	使用Gecko的产品名。可能是Netscape、Firefox等
应用程序或产品版本号	否	应用程序或产品的版本号; 用于区分Mozilla版本号和Gecko版本号



为了帮助读者更好地理解 Gecko 的用户代理字符串,下面我们来看几个从基于 Gecko 的浏览器中取得的字符串。

Windows XP 下的 Netscape 6.2.1:

```
Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:0.9.4) Gecko/20011128 Netscape6/6.2.1
```

Linux 下的 SeaMonkey 1.1a:

```
Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1b2) Gecko/20060823 SeaMonkey/1.1a
```

Windows XP 下的 Firefox 2.0.0.11:

```
Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.11) Gecko/20071127 Firefox/2.0.0.11
```

Mac OS X 下的 Camino 1.5.1:

```
Mozilla/5.0 (Macintosh; U; Intel Mac OS X; en; rv:1.8.1.6) Gecko/20070809 Camino/1.5.1
```

以上这些用户代理字符串都取自基于 Gecko 的浏览器(只是版本有所不同)。很多时候,检测特定的浏览器还不如搞清楚它是否基于 Gecko 更重要。每个字符串中的 Mozilla 版本都是 5.0,自从第一个基于 Gecko 的浏览器发布时修改成这个样子,至今就没有改变过;而且,看起来以后似乎也不会有什么变化。

随着 Firefox 4 发布, Mozilla 简化了这个用户代理字符串。主要改变包括以下几方面。

- 删除了“语言”记号(例如,前面例子中的“en-US”)。
- 在浏览器使用强加密(默认设置)时,不显示“加密类型”。也就是说, Mozilla 用户代理字符串中不会再出现“U”,而“I”和“N”还会照常出现。
- “平台”记号从 Windows 用户代理字符串中删除了,“操作系统或 CPU”中始终都包含“Windows”字符串。
- “Gecko 版本号”固定为“Gecko/20100101”。

最后, Firefox 4 用户代理字符串变成了下面这个样子:

```
Mozilla/5.0 (Windows NT 6.1; rv:2.0.1) Gecko/20100101 Firefox 4.0.1
```

## 5. WebKit

2003 年, Apple 公司宣布要发布自己的 Web 浏览器,名字定为 Safari。Safari 的呈现引擎叫 WebKit,是 Linux 平台中 Konqueror 浏览器的呈现引擎 KHTML 的一个分支。几年后, WebKit 独立出来成为了一个开源项目,专注于呈现引擎的开发。

这款新浏览器和呈现引擎的开发人员也遇到了与 Internet Explorer 3.0 类似的问题:如何确保这款浏览器不被流行的站点拒之门外?答案就是向用户代理字符串中放入足够多的信息,以便站点能够信任它与其他流行的浏览器是兼容的。于是, WebKit 的用户代理字符串就具备了如下格式:

```
Mozilla/5.0 (平台; 加密类型; 操作系统或 CPU; 语言) AppleWebKit/AppleWebKit 版本号  
(KHTML, like Gecko) Safari/Safari 版本号
```

以下就是一个示例:

```
Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en) AppleWebKit/124 (KHTML, like Gecko)  
Safari/125.1
```

显然,这又是一个很长的用户代理字符串。其中不仅包含了 Apple WebKit 的版本号,也包含了 Safari 的版本号。出于兼容性的考虑,有关人员很快就决定了将 Safari 标识为 Mozilla。至今,基于 WebKit 的

所有浏览器都将自己标识为 Mozilla 5.0，与基于 Gecko 的浏览器完全一样。但 Safari 的版本号则通常是浏览器的编译版本号，不一定与发布时的版本号对应。换句话说，虽然 Safari 1.25 的用户代理字符串中包含数字 125.1，但两者却不——对应。

Safari 预发行 1.0 版用户代理字符串中最耐人寻味，也是最饱受诟病的部分就是字符串“(KHTML, like Gecko)”。Apple 因此收到许多开发人员的反馈，他们认为这个字符串明显是在欺骗客户端和服务端，实际上是想让它们把 Safari 当成 Gecko（好像光添加 Mozilla/5.0 还嫌不够）。Apple 的回应与微软在 IE 的用户代理字符串遭到责难时如出一辙：Safari 与 Mozilla 兼容，因此网站不应该将 Safari 用户拒之门外，否则用户就会认为自己的浏览器不受支持。

到了 Safari 3.0 发布时，其用户代理字符串又稍微变长了一点。下面这个新增的 Version 记号一直到现在都被用来标识 Safari 实际的版本号：

```
Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en) AppleWebKit/522.15.5 (KHTML, like Gecko) Version/3.0.3 Safari/522.15.5
```

需要注意的是，这个变化只在 Safari 中有，在 WebKit 中没有。换句话说，其他基于 WebKit 的浏览器可能没有这个变化。一般来说，确定浏览器是否基于 WebKit 要比确定它是不是 Safari 更有价值，就像针对 Gecko 一样。

## 6. Konqueror

与 KDE Linux 集成的 Konqueror，是一款基于 KHTML 开源呈现引擎的浏览器。尽管 Konqueror 只能在 Linux 中使用，但它也有数量可观的用户。为确保最大限度的兼容性，Konqueror 效仿 IE 选择了如下用户代理字符串格式：

```
Mozilla/5.0 (compatible; Konqueror/ 版本号; 操作系统或 CPU )
```

不过，为了与 WebKit 的用户代理字符串的变化保持一致，Konqueror 3.2 又有了变化，以如下格式将自己标识为 KHTML：

```
Mozilla/5.0 (compatible; Konqueror/ 版本号; 操作系统或 CPU) KHTML/ KHTML 版本号 (like Gecko)
```

下面是一个例子：

```
Mozilla/5.0 (compatible; Konqueror/3.5; SunOS) KHTML/3.5.0 (like Gecko)
```

其中，Konqueror 与 KHTML 的版本号比较一致，即使有差别也很小，例如 Konqueror 3.5 使用 KHTML 3.5.1。

## 7. Chrome

谷歌公司的 Chrome 浏览器以 WebKit 作为呈现引擎，但使用了不同的 JavaScript 引擎。在 Chrome 0.2 这个最初的 beta 版中，用户代理字符串完全取自 WebKit，只添加了一段表示 Chrome 版本号的信息，格式如下：

```
Mozilla/5.0 ( 平台; 加密类型; 操作系统或 CPU; 语言) AppleWebKit/AppleWebKit 版本号 (KHTML, like Gecko) Chrome/ Chrome 版本号 Safari/ Safari 版本
```

Chrome 7 的完整的用户代理字符串如下：

```
Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) AppleWebKit/534.7 (KHTML, like Gecko) Chrome/7.0.517.44 Safari/534.7
```

其中，WebKit 版本与 Safari 版本看起来似乎始终会保持一致，尽管没有十分的把握。

## 8. Opera

仅就用户代理字符串而言，Opera 应该是最有争议的一款浏览器了。Opera 默认的用户代理字符串是所有现代浏览器中最合理的——正确地标识了自身及其版本号。在 Opera 8.0 之前，其用户代理字符串采用如下格式：

Opera/ 版本号 (操作系统或 CPU; 加密类型) [语言]

Windows XP 中的 Opera 7.54 会显示下面的用户代理字符串：

Opera/7.54 (Windows NT 5.1; U) [en]

Opera 8 发布后，用户代理字符串的“语言”部分被移到圆括号内，以便更好地与其他浏览器匹配，如下所示：

Opera/ 版本号 (操作系统或 CPU; 加密类型; 语言)

Windows XP 中的 Opera 8 会显示下面的用户代理字符串：

Opera/8.0 (Windows NT 5.1; U; en)

默认情况下，Opera 会以上面这种简单的格式返回一个用户代理字符串。目前来看，Opera 也是主要浏览器中唯一一个使用产品名和版本号来完全彻底地标识自身的浏览器。可是，与其他浏览器一样，Opera 在使用自己的用户代理字符串时也遇到了问题。即使技术上正确，但因特网上仍然有不少浏览器嗅探代码，只钟情于报告 Mozilla 产品名的那些用户代理字符串。另外还有相当数量的代码则只对 IE 或 Gecko 感兴趣。Opera 没有选择通过修改自身的用户代理字符串来迷惑嗅探代码，而是干脆选择通过修改自身的用户代理字符串将自身标识为一个完全不同的浏览器。

Opera 9 以后，出现了两种修改用户代理字符串的方式。一种方式是将自身标识为另外一个浏览器，如 Firefox 或者 IE。在这种方式下，用户代理字符串就如同 Firefox 或 IE 的用户代理字符串一样，只不过末尾追加了字符串 Opera 及 Opera 的版本号。下面是一个例子：

Mozilla/5.0 (Windows NT 5.1; U; en; rv:1.8.1) Gecko/20061208 Firefox/2.0.0 Opera 9.50

Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; en) Opera 9.50

第一个字符串将 Opera 9.5 标识为 Firefox 2，同时带有 Opera 版本信息。第二个字符串将 Opera 9.5 标识为 IE6，也包含了 Opera 版本信息。这两个用户代理字符串可以通过针对 Firefox 或 IE 的大多数测试，不过还是为识别 Opera 留下了余地。

Opera 标识自身的另一种方式，就是把自己装扮成 Firefox 或 IE。在这种隐瞒真实身份的情况下，用户代理字符串实际上与其他浏览器返回的相同——既没有 Opera 字样，也不包含 Opera 版本信息。换句话说，在启用了身份隐瞒功能的情况下，无法将 Opera 和其他浏览器区别开来。另外，由于 Opera 喜欢在不告知用户的情况下针对站点来设置用户代理字符串，因此问题就更复杂化了。例如，打开 My Yahoo! 站点 (<http://my.yahoo.com>) 会自动导致 Opera 将自己装扮成 Firefox。如此一来，要想识别 Opera 就难上加难了。



在 Opera 7 以前的版本中，Opera 会解析 Windows 操作系统字符串的含义。例如，Windows NT 5.1 实际上就是 Windows XP，因此 Opera 会在用户代理字符串中包含 Windows XP 而非 Windows NT 5.1。为了与其他浏览器更兼容，Opera 7 开始包含正式的操作系统版本，而非解析后的版本。

Opera 10 对代理字符串进行了修改。现在的格式是：

Opera/9.80 (操作系统或 CPU; 加密类型; 语言) Presto/Presto 版本号 Version/版本号

注意，初始的版本号 Opera/9.80 是固定不变的。实际并没有 Opera 9.8，但工程师们担心写得不好的浏览器嗅探脚本会将 Opera/10.0 错误的解释为 Opera 1，而不是 Opera 10。因此，Opera 10 又增加了 Presto 记号（Presto 是 Opera 的呈现引擎）和 Version 记号，后者用以保存实际的版本号。以下是 Windows7 中 Opera 10.63 的用户代理字符串：

Opera/9.80 (Windows NT 6.1; U; en) Presto/2.6.30 Version/10.63

## 9. iOS 和 Android

移动操作系统 iOS 和 Android 默认的浏览器都基于 WebKit，而且都像它们的桌面版一样，共享相同的基本用户代理字符串格式。iOS 设备的基本格式如下：

Mozilla/5.0 (平台; 加密类型; 操作系统或 CPU like Mac OS X; 语言)  
AppleWebKit/AppleWebKit 版本号 (KHTML, like Gecko) Version/浏览器版本号  
Mobile/移动版本号 Safari/Safari 版本号

注意用于辅助确定 Mac 操作系统的“like Mac OS X”和额外的 Mobile 记号。一般来说，Mobile 记号的版本号（移动版本号）没什么用，主要是用来确定 WebKit 是移动版，而非桌面版。而平台则可能是“iPhone”、“iPod”或“iPad”。例如：

Mozilla/5.0 (iPhone; U; CPU iPhone OS 3\_0 like Mac OS X; en-us)  
AppleWebKit/528.18 (KHTML, like Gecko) Version/4.0 Mobile/7A341 Safari/528.16

在 iOS 3 之前，用户代理字符串中不会出现操作系统版本号。

Android 浏览器中的默认格式与 iOS 的格式相似，没有移动版本号（但有 Mobile 记号）。例如：

Mozilla/5.0 (Linux; U; Android 2.2; en-us; Nexus One Build/FRF91)  
AppleWebKit/533.1 (KHTML, like Gecko) Version/4.0 Mobile Safari/533.1

这是 Google Nexus One 手机的用户代理字符串。不过，其他 Android 设备的模式也一样。

## 9.3.2 用户代理字符串检测技术

考虑到历史原因以及现代浏览器中用户代理字符串的使用方式，通过用户代理字符串来检测特定的浏览器并不是一件轻松的事。因此，首先要确定的往往是你需要多么具体的浏览器信息。一般情况下，知道呈现引擎和最低限度的版本就足以决定正确的操作方法了。例如，我们不推荐使用下列代码：

```
if (isIE6 || isIE7) { //不推荐!!!  
    //代码  
}
```

这个例子是想要在浏览器为 IE6 或 IE7 时执行相应代码。这种代码其实是很脆弱的，因为它要依据特定的版本来决定做什么。如果是 IE8 怎么办呢？只要 IE 有新版本出来，就必须更新这些代码。不过，像下面这样使用相对版本号则可以避免此问题：

```
if (ieVer >=6){  
    //代码  
}
```

这个例子首先检测 IE 的版本号是否至少等于 6，如果是则执行相应操作。这样就可以确保相应的代码将来照样能够起作用。我们下面的浏览器检测脚本就将本着这种思路来编写。

## 1. 识别呈现引擎

如前所述，确切知道浏览器的名字和版本号不如确切知道它使用的是哪种呈现引擎。如果 Firefox、Camino 和 Netscape 都使用相同版本的 Gecko，那它们一定支持相同的特性。类似地，不管是什么浏览器，只要它跟 Safari 3 使用的是同一个版本的 WebKit，那么该浏览器也就跟 Safari 3 具备同样的功能。因此，我们要编写的脚本将主要检测五大呈现引擎：IE、Gecko、WebKit、KHTML 和 Opera。

为了不在全局作用域中添加多余的变量，我们将使用模块增强模式来封装检测脚本。检测脚本的基本代码结构如下所示：

```
var client = function(){

    var engine = {

        //呈现引擎
        ie: 0,
        gecko: 0,
        webkit: 0,
        khtml: 0,
        opera: 0,

        //具体的版本号
        ver: null

    };

    //在此检测呈现引擎、平台和设备

    return {
        engine : engine
    };

}();
```

这里声明了一个名为 client 的全局变量，用于保存相关信息。匿名函数内部定义了一个局部变量 engine，它是一个包含默认设置的对象字面量。在这个对象字面量中，每个呈现引擎都对应着一个属性，属性的值默认为 0。如果检测到了哪个呈现引擎，那么就以浮点数值形式将该引擎的版本号写入相应的属性。而呈现引擎的完整版本（是一个字符串），则被写入 ver 属性。作这样的区分可以支持像下面这样编写代码：

```
if (client.engine.ie) { //如果是 IE, client.ie 的值应该大于 0
    //针对 IE 的代码
} else if (client.engine.gecko > 1.5){
    if (client.engine.ver == "1.8.1"){
        //针对这个版本执行某些操作
    }
}
```

在检测到一个呈现引擎之后，其 client.engine 中对应的属性将被设置为一个大于 0 的值，该值可以转换成布尔值 true。这样，就可以在 if 语句中检测相应的属性，以确定当前使用的呈现引擎，连具体的版本号都不必考虑。鉴于每个属性都包含一个浮点数值，因此有可能丢失某些版本信息。例如，将字符串“1.8.1”传入 parseFloat() 后会得到数值 1.8。不过，在必要的时候可以检测 ver 属性，该属性中会保存完整的版本信息。

要正确地识别呈现引擎，关键是检测顺序要正确。由于用户代理字符串存在诸多不一致的地方，如果检测顺序不对，很可能导致检测结果不正确。为此，第一步就是识别 Opera，因为它的用户代理字

字符串有可能完全模仿其他浏览器。我们不相信 Opera，是因为（任何情况下）其用户代理字符串（都）不会将自己标识为 Opera。

要识别 Opera，必须得检测 window.opera 对象。Opera 5 及更高版本中都有这个对象，用以保存与浏览器相关的标识信息以及与浏览器直接交互。在 Opera 7.6 及更高版本中，调用 version() 方法可以返回一个表示浏览器版本的字符串，而这也是确定 Opera 版本号的最佳方式。要检测更早版本的 Opera，可以直接检查用户代理字符串，因为那些版本还不支持隐瞒身份。不过，2007 底 Opera 的最高版本已经是 9.5 了，所以不太可能有人还在使用 7.6 之前的版本。那么，检测呈现引擎代码的第一步，就是编写如下代码：

```
if (window.opera){
    engine.ver = window.opera.version();
    engine.opera = parseFloat(engine.ver);
}
```

这里，将版本的字符串表示保存在了 engine.ver 中，将浮点数值表示的版本保存在了 engine.opera 中。如果浏览器是 Opera，测试 window.opera 就会返回 true；否则，就要看看是其他的什么浏览器了。

应该放在第二位检测的呈现引擎是 WebKit。因为 WebKit 的用户代理字符串中包含 "Gecko" 和 "KHTML" 这两个子字符串，所以如果首先检测它们，很可能会得出错误的结论。

不过，WebKit 的用户代理字符串中的 "AppleWebKit" 是独一无二的，因此检测这个字符串最合适。下面就是检测该字符串的示例代码：

```
var ua = navigator.userAgent;

if (window.opera){
    engine.ver = window.opera.version();
    engine.opera = parseFloat(engine.ver);
} else if (/AppleWebKit\/(\S+)/.test(ua)){
    engine.ver = RegExp["$1"];
    engine.webkit = parseFloat(engine.ver);
}
```

代码首先将用户代理字符串保存在变量 ua 中。然后通过正则表达式来测试其中是否包含字符串 "AppleWebKit"，并使用捕获组来取得版本号。由于实际的版本号中可能会包含数字、小数点和字母，所以捕获组中使用了表示非空格的特殊字符 (\S)。用户代理字符串中的版本号与下一部分的分隔符是一个空格，因此这个模式可以保证捕获所有版本信息。test() 方法基于用户代理字符串运行正则表达式。如果返回 true，就将捕获的版本号保存在 engine.ver 中，而将版本号的浮点表示保存在 engine.webkit 中。WebKit 版本与 Safari 版本的详细对应情况如下表所示。

Safari版本号	最低限度的WebKit版本号	Safari版本号	最低限度的WebKit版本号
1.0至1.0.2	85.7	1.3	312.1
1.0.3	85.8.2	1.3.1	312.5
1.1至1.1.1	100	1.3.2	312.8
1.2.2	125.2	2.0	412
1.2.3	125.4	2.0.1	412.7
1.2.4	125.5.5	2.0.2	416.11

Safari版本号	最低限度的WebKit版本号	Safari版本号	最低限度的WebKit版本号
2.0.3	417.9	3.0.4	523.10
2.0.4	418.8	3.1	525



有时候, Safari 版本并不会与 WebKit 版本严格地一一对应, 也可能会存在某些小版本上的差异。这个表中只是列出了最可能的 WebKit 版本, 但不保证精确。

接下来要测试的呈现引擎是 KHTML。同样, KHTML 的用户代理字符串中也包含 "Gecko", 因此在排除 KHTML 之前, 我们无法准确检测基于 Gecko 的浏览器。KHTML 的版本号与 WebKit 的版本号在用户代理字符串中的格式差不多, 因此可以使用类似的正则表达式。此外, 由于 Konqueror 3.1 及早版本中不包含 KHTML 的版本, 故而就要使用 Konqueror 的版本来代替。下面就是相应的检测代码。

```
var ua = navigator.userAgent;

if (window.opera){
    engine.ver = window.opera.version();
    engine.opera = parseFloat(engine.ver);
} else if (/AppleWebKit\/(\S+)/.test(ua)){
    engine.ver = RegExp["$1"];
    engine.webkit = parseFloat(engine.ver);
} else if (/KHTML\/(\S+)/.test(ua) || /Konqueror\/([^\;]+)/.test(ua)){
    engine.ver = RegExp["$1"];
    engine.khtml = parseFloat(engine.ver);
}
```

与前面一样, 由于 KHTML 的版本号与后继的标记之间有一个空格, 因此仍然要使用特殊的非空格字符来取得与版本有关的所有字符。然后, 将字符串形式的版本信息保存在 engine.ver 中, 将浮点数值形式的版本保存在 engine.khtml 中。如果 KHTML 不在用户代理字符串中, 那么就要匹配 Konqueror 后跟一个斜杠, 再后跟不包含分号的所有字符。

在排除了 WebKit 和 KHTML 之后, 就可以准确地检测 Gecko 了。但是, 在用户代理字符串中, Gecko 的版本号不会出现在字符串 "Gecko" 的后面, 而是会出现在字符串 "rv:" 的后面。这样, 我们就必须使用一个比前面复杂一些的正则表达式, 如下所示。

```
var ua = navigator.userAgent;

if (window.opera){
    engine.ver = window.opera.version();
    engine.opera = parseFloat(engine.ver);
} else if (/AppleWebKit\/(\S+)/.test(ua)){
    engine.ver = RegExp["$1"];
    engine.webkit = parseFloat(engine.ver);
} else if (/KHTML\/(\S+)/.test(ua)) {
    engine.ver = RegExp["$1"];
    engine.khtml = parseFloat(engine.ver);
} else if (/rv:([^\;]+)\) Gecko\/\d{8}/.test(ua)){
    engine.ver = RegExp["$1"];
    engine.gecko = parseFloat(engine.ver);
}
```

Gecko 的版本号位于字符串"rv:"与一个闭括号之间,因此为了提取出这个版本号,正则表达式要查找所有不是闭括号的字符,还要查找字符串"Gecko/"后跟 8 个数字。如果上述模式匹配,就提取出版本号并将其保存在相应的属性中。Gecko 版本号与 Firefox 版本号的对应关系如下表所示。

Firefox版本号	最低限度的Gecko版本号	Firefox版本号	最低限度的Gecko版本号
1.0	1.7.5	3.5	1.9.1
1.5	1.8.0	3.6	1.9.2
2.0	1.8.1	4.0	2.0.0
3.0	1.9.0		



与 Safari 跟 WebKit 一样,Firefox 与 Gecko 的版本号也不一定严格对应。

最后一个要检测的呈现引擎就是 IE 了。IE 的版本号位于字符串"MSIE"的后面、一个分号的前面,因此相应的正则表达式非常简单,如下所示:

```
var ua = navigator.userAgent;

if (window.opera){
    engine.ver = window.opera.version();
    engine.opera = parseFloat(engine.ver);
} else if (/AppleWebKit\/(\S+)/.test(ua)){
    engine.ver = RegExp["$1"];
    engine.webkit = parseFloat(engine.ver);
} else if (/KHTML\/(\S+)/.test(ua)) {
    engine.ver = RegExp["$1"];
    engine.khtml = parseFloat(engine.ver);
} else if (/rv:([^\s]+)\ Gecko\/\d{8}/.test(ua)){
    engine.ver = RegExp["$1"];
    engine.gecko = parseFloat(engine.ver);
} else if (/MSIE ([^;]+)/.test(ua)){
    engine.ver = RegExp["$1"];
    engine.ie = parseFloat(engine.ver);
}
```

以上呈现引擎检测脚本的最后一部分,就是在正则表达式中使用取反的字符类来取得不是分号的所有字符。IE 通常会保证以标准浮点数值形式给出其版本号,但有时候也不一定。因此,取反的字符类[^;]可以确保取得多个小数点以及任何可能的字符。

## 2. 识别浏览器

大多数情况下,识别了浏览器的呈现引擎就足以为我们采取正确的操作提供依据了。可是,只有呈现引擎还不能说明存在所需的 JavaScript 功能。苹果公司的 Safari 浏览器和谷歌公司的 Chrome 浏览器都使用 WebKit 作为呈现引擎,但它们的 JavaScript 引擎却不一样。在这两款浏览器中,client.webkit 都会返回非 0 值,但仅知道这一点恐怕还不够。对于它们,有必要像下面这样为 client 对象再添加一些新的属性。

```
var client = function(){
    var engine = {
```



```

        //呈现引擎
        ie: 0,
        gecko: 0,
        webkit: 0,
        khtml: 0,
        opera: 0,

        //具体的版本
        ver: null
    };

    var browser = {

        //浏览器
        ie: 0,
        firefox: 0,
        safari: 0,
        konq: 0,
        opera: 0,
        chrome: 0,

        //具体的版本
        ver: null
    };

    //在此检测呈现引擎、平台和设备

    return {
        engine: engine,
        browser: browser
    };

}();

```

代码中又添加了私有变量 `browser`，用于保存每个主要浏览器的属性。与 `engine` 变量一样，除了当前使用的浏览器，其他属性的值将保持为 0；如果是当前使用的浏览器，则这个属性中保存的是浮点数值形式的版本号。同样，`ver` 属性中在必要时将会包含字符串形式的浏览器完整版本号。由于大多数浏览器与其呈现引擎密切相关，所以下面示例中检测浏览器的代码与检测呈现引擎的代码是混合在一起的。

```

//检测呈现引擎及浏览器
var ua = navigator.userAgent;
if (window.opera){
    engine.ver = browser.ver = window.opera.version();
    engine.opera = browser.opera = parseFloat(engine.ver);
} else if (/AppleWebKit\/(\S+)/.test(ua)){
    engine.ver = RegExp["$1"];
    engine.webkit = parseFloat(engine.ver);

    //确定是 Chrome 还是 Safari
    if (/Chrome\/(\S+)/.test(ua)){
        browser.ver = RegExp["$1"];
        browser.chrome = parseFloat(browser.ver);
    } else if (/Version\/(\S+)/.test(ua)){
        browser.ver = RegExp["$1"];
        browser.safari = parseFloat(browser.ver);
    } else {
        //近似地确定版本号
        var safariVersion = 1;
    }
}

```

```

        if (engine.webkit < 100){
            safariVersion = 1;
        } else if (engine.webkit < 312){
            safariVersion = 1.2;
        } else if (engine.webkit < 412){
            safariVersion = 1.3;
        } else {
            safariVersion = 2;
        }

        browser.safari = browser.ver = safariVersion;
    }
} else if (/KHTML\/(\S+)/.test(ua) || /Konqueror\/([^\;]+)/.test(ua)){
    engine.ver = browser.ver = RegExp["$1"];
    engine.khtml = browser.konq = parseFloat(engine.ver);
} else if (/rv:([^\)]+)\) Gecko\/\d{8}/.test(ua)){
    engine.ver = RegExp["$1"];
    engine.gecko = parseFloat(engine.ver);

    //确定是不是 Firefox
    if (/Firefox\/(\S+)/.test(ua)){
        browser.ver = RegExp["$1"];
        browser.firefox = parseFloat(browser.ver);
    }
} else if (/MSIE ([^\;]+)/.test(ua)){
    engine.ver = browser.ver = RegExp["$1"];
    engine.ie = browser.ie = parseFloat(engine.ver);
}
}

```

对 Opera 和 IE 而言, browser 对象中的值等于 engine 对象中的值。对 Konqueror 而言, browser.konq 和 browser.ver 属性分别等于 engine.khtml 和 engine.ver 属性。

为了检测 Chrome 和 Safari, 我们在检测引擎的代码中添加了 if 语句。提取 Chrome 的版本号时, 需要查找字符串 "Chrome/" 并取得该字符串后面的数值。而提取 Safari 的版本号时, 则需要查找字符串 "Version/" 并取得其后的数值。由于这种方式仅适用于 Safari 3 及更高版本, 因此需要一些备用的代码, 将 WebKit 的版本号近似地映射为 Safari 的版本号 (参见上一小节中的表格)。

在检测 Firefox 的版本时, 首先要找到字符串 "Firefox/", 然后提取出该字符串后面的数值 (即版本号)。当然, 只有呈现引擎被判别为 Gecko 时才会这样做。

有了上面这些代码之后, 我们就可以编写下面的逻辑。

```

if (client.engine.webkit) { //if it's WebKit
    if (client.browser.chrome){
        //执行针对 Chrome 的代码
    } else if (client.browser.safari){
        //执行针对 Safari 的代码
    }
} else if (client.engine.gecko){
    if (client.browser.firefox){
        //执行针对 Firefox 的代码
    } else {
        //执行针对其他 Gecko 浏览器的代码
    }
}
}

```

### 3. 识别平台

很多时候，只要知道呈现引擎就足以编写出适当的代码了。但在某些条件下，平台可能是必须关注的问题。那些具有各种平台版本的浏览器（如 Safari、Firefox 和 Opera）在不同的平台下可能会有不同的问题。目前的三大主流平台是 Windows、Mac 和 Unix（包括各种 Linux）。为了检测这些平台，还需要像下面这样再添加一个新对象。

```
var client = function(){  
    var engine = {  
        //呈现引擎  
        ie: 0,  
        gecko: 0,  
        webkit: 0,  
        khtml: 0,  
        opera: 0,  
  
        //具体的版本号  
        ver: null  
    };  
  
    var browser = {  
        //浏览器  
        ie: 0,  
        firefox: 0,  
        safari: 0,  
        konq: 0,  
        opera: 0,  
        chrome: 0,  
  
        //具体的版本号  
        ver: null  
    };  
  
    var system = {  
        win: false,  
        mac: false,  
        x11: false  
    };  
  
    //在此检测呈现引擎、平台和设备  
  
    return {  
        engine: engine,  
        browser: browser,  
        system: system  
    };  
  
}();
```

显然，上面的代码中又添加了一个包含 3 个属性的新变量 `system`。其中，`win` 属性表示是否为 Windows 平台，`mac` 表示 Mac，而 `x11` 表示 Unix。与呈现引擎不同，在不能访问操作系统或版本的情况下，平台信息通常是很有限制的。对这三个平台而言，浏览器一般只报告 Windows 版本。为此，新变量 `system` 的每个属性最初都保存着布尔值 `false`，而不是像呈现引擎属性那样保存着数字值。

在确定平台时，检测 `navigator.platform` 要比检测用户代理字符串更简单，后者在不同浏览器中会给出不同的平台信息。而 `navigator.platform` 属性可能的值包括 "Win32"、"Win64"、

"MacPPC"、"MacIntel"、"X11"和"Linux i686"，这些值在不同的浏览器中都是一致的。检测平台的代码非常直观，如下所示：

```
var p = navigator.platform;
system.win = p.indexOf("Win") == 0;
system.mac = p.indexOf("Mac") == 0;
system.x11 = (p.indexOf("X11") == 0) || (p.indexOf("Linux") == 0);
```

以上代码使用 `indexOf()` 方法来查找平台字符串的开始位置。虽然"Win32"是当前浏览器唯一支持的 Windows 字符串，但随着向 64 位 Windows 架构的迁移，将来很可能会出现"Win64"平台信息值。为了对此有所准备，检测平台的代码中查找的只是字符串"Win"的开始位置。而检测 Mac 平台的方式也类似，同样是考虑到了 MacPPC 和 MacIntel。在检测 Unix 时，则同时检查了字符串"X11"和"Linux"在平台字符串中的开始位置，从而确保了代码能够向前兼容其他变体。



Gecko 的早期版本在所有 Windows 平台中都返回字符串"Windows"，在所有 Mac 平台中则都返回字符串"Macintosh"。不过，这都是 Firefox 1 发布以前的事了，Firefox 1 确定了 `navigator.platform` 的值。

#### 4. 识别 Windows 操作系统

在 Windows 平台下，还可以从用户代理字符串中进一步取得具体的操作系统信息。在 Windows XP 之前，Windows 有两种版本，分别针对家庭用户和商业用户。针对家庭用户的版本分别是 Windows 95、98 和 Windows ME。而针对商业用户的版本则一直叫做 Window NT，最后由于市场原因改名为 Windows 2000。这两个产品线后来又合并成一个由 Windows NT 发展而来的公共的代码基，代表产品就是 Windows XP。随后，微软在 Windows XP 基础上又构建了 Windows Vista。

只有了解这些信息，才能搞清楚用户代理字符串中 Windows 操作系统的具体版本。下表列出了不同浏览器在表示不同的 Windows 操作系统时给出的不同字符串。

Windows版本	IE 4+	Gecko	Opera < 7	Opera 7+	WebKit
95	"Windows 95"	"Win95"	"Windows 95"	"Windows 95"	n/a
98	"Windows 98"	"Win98"	"Windows 98"	"Windows 98"	n/a
NT 4.0	"Windows NT"	"WinNT4.0"	"Windows NT 4.0"	"Windows NT 4.0"	n/a
2000	"Windows NT 5.0"	"Windows NT 5.0"	"Windows 2000"	"Windows NT 5.0"	n/a
ME	"Win 9x 4.90"	"Win 9x 4.90"	"Windows ME"	"Win 9x 4.90"	n/a
XP	"Windows NT 5.1"	"Windows NT 5.1"	"Windows XP"	"Windows NT 5.1"	"Windows NT 5.1"
Vista	"Windows NT 6.0"	"Windows NT 6.0"	n/a	"Windows NT 6.0"	"Windows NT 6.0"
7	"Windows NT 6.1"	"Windows NT 6.1"	n/a	"Windows NT 6.1"	"Windows NT 6.1"

由于用户代理字符串中的 Windows 操作系统版本表示方法各异，因此检测代码并不十分直观。好在，从 Windows 2000 开始，表示操作系统的字符串大部分都还相同，只有版本号有变化。为了检测不同的 Windows 操作系统，必须要使用正则表达式。由于使用 Opera 7 之前版本的用户已经不多了，因此我们可以忽略这部分浏览器。

第一步就是匹配 Windows 95 和 Windows 98 这两个字符串。对这两个字符串，只有 Gecko 与其他浏览器不同，即没有"dows"，而且"Win"与版本号之间没有空格。要匹配这个模式，可以使用下面这个简单的正则表达式。

```
/Win(?:dows )?{[^do]{2}}/
```

这个正则表达式中的捕获组会返回操作系统的版本。由于版本可能是任何两个字符编码（例如 95、98、9x、NT、ME 及 XP），因此要使用两个非空格字符。

Gecko 在表示 Windows NT 时会在末尾添加"4.0"，与其查找实际的字符串，不如像下面这样查找小数值更合适。

```
/Win(?:dows )?{[^do]{2}}{(\d+\.\d+)?}/
```

这样，正则表达式中就包含了第二个捕获组，用于取得 NT 的版本号。由于该版本号对于 Windows 95 和 Windows 98 而言是不存在的，所以必须设置为可选。这个模式与 Opera 表示 Windows NT 的字符串之间唯一的区别，就是"NT"与"4.0"之间的空格，这在模式中很容易添加。

```
/Win(?:dows )?{[^do]{2}}\s?(\d+\.\d+)?/
```

经过一番修改之后，这个正则表达式也可以成功地匹配 Windows ME、Windows XP 和 Windows Vista 的字符串了。具体来说，第一个捕获组将会匹配 95、98、9x、NT、ME 或 XP。第二个捕获组则只针对 Windows ME 及所有 Windows NT 的变体。这个信息可以作为具体的操作系统信息保存在 system.win 属性中，如下所示。

```
if (system.win){
    if (/Win(?:dows )?{[^do]{2}}\s?(\d+\.\d+)?/.test(ua)){
        if (RegExp["$1"] == "NT"){
            switch(RegExp["$2"]){
                case "5.0":
                    system.win = "2000";
                    break;
                case "5.1":
                    system.win = "XP";
                    break;
                case "6.0":
                    system.win = "Vista";
                    break;
                case "6.1":
                    system.win = "7";
                    break;
                default:
                    system.win = "NT";
                    break;
            }
        } else if (RegExp["$1"] == "9x"){
            system.win = "ME";
        } else {
            system.win = RegExp["$1"];
        }
    }
}
```

如果 `system.win` 的值为 `true`,那么就使用这个正则表达式从用户代理字符串中提取具体的信息。鉴于 Windows 将来的某个版本也许不能使用这个方法检测,所以第一步应该先检测用户代理字符串是否与这个模式匹配。在模式匹配的情况下,第一个捕获组中可能会包含"95"、"98"、"9x"或"NT"。如果这个值是"NT",可以将 `system.win` 设置为相应操作系统的字符串;如果是"9x",那么 `system.win` 就要设置成"ME";如果是其他值,则将所捕获的值直接赋给 `system.win`。有了这些检测平台的代码后,我们就可以编写如下代码。

```
if (client.system.win){
    if (client.system.win == "XP") {
        //说明是XP
    } else if (client.system.win == "Vista"){
        //说明是Vista
    }
}
```

由于非空字符串会转换为布尔值 `true`,因此可以将 `client.system.win` 作为布尔值用在 `if` 语句中。而在需要更多有关操作系统的信息时,则可以使用其中保存的字符串值。

## 5. 识别移动设备

2006 年到 2007 年,移动设备中 Web 浏览器的应用呈爆炸性增长。四大主要浏览器都推出了手机版和在其他设备中运行的版本。要检测相应的设备,第一步是为要检测的所有移动设备添加属性,如下所示。

```
var client = function(){

    var engine = {

        //呈现引擎
        ie: 0,
        gecko: 0,
        webkit: 0,
        khtml: 0,
        opera: 0,

        //具体的版本号
        ver: null
    };

    var browser = {

        //浏览器
        ie: 0,
        firefox: 0,
        safari: 0,
        konq: 0,
        opera: 0,
        chrome: 0,

        //具体的版本号
        ver: null
    };

    var system = {
        win: false,
        mac: false,
```

```

xll: false,

//移动设备
iphone: false,
ipod: false,
ipad: false,
ios: false,
android: false,
nokiaN: false,
winMobile: false    };

//在此检测呈现引擎、平台和设备

return {
    engine: engine,
    browser: browser,
    system: system
};

}();

```

然后，通常简单地检测字符串"iPhone"、"iPod"和"iPad"，就可以分别设置相应属性的值了。

```

system.iphone = ua.indexOf("iPhone") > -1;
system.ipod = ua.indexOf("iPod") > -1;
system.ipad = ua.indexOf("iPad") > -1;

```

除了知道 iOS 设备，最好还能知道 iOS 的版本号。在 iOS 3 之前，用户代理字符串中只包含"CPU like Mac OS"，后来 iPhone 中又改成"CPU iPhone OS 3\_0 like Mac OS X"，iPad 中又改成"CPU OS 3\_2 like Mac OS X"。也就是说，检测 iOS 需要正则表达式反映这些变化。

```

//检测 iOS 版本
if (system.mac && ua.indexOf("Mobile") > -1){
    if (/CPU (?:iPhone )?OS (\d+_\d+)/.test(ua)){
        system.ios = parseFloat(RegExp.$1.replace("_", "."));
    } else {
        system.ios = 2; //不能真正检测出来，所以只能猜测
    }
}

```

检查系统是不是 Mac OS、字符串中是否存在"Mobile"，可以保证无论是什么版本，system.ios 中都不会是 0。然后，再使用正则表达式确定是否存在 iOS 的版本号。如果有，将 system.ios 设置为表示版本号的浮点值；否则，将版本设置为 2。（因为没有办法确定到底是什么版本，所以设置为更早的版本比较稳妥。）

检测 Android 操作系统也很简单，也就是搜索字符串"Android"并取得紧随其后的版本号。

```

//检测 Android 版本
if (/Android (\d+\.?\d+)/.test(ua)){
    system.android = parseFloat(RegExp.$1);
}

```

由于所有版本的 Android 都有版本值，因此这个正则表达式可以精确地检测所有版本，并将 system.android 设置为正确的值。

诺基亚 N 系列手机使用的也是 WebKit，其用户代理字符串与其他基于 WebKit 的手机很相似，例如：

```

Mozilla/5.0 (SymbianOS/9.2; U; Series60/3.1 NokiaN95/11.0.026; Profile MIDP-2.0
Configuration/CLDC-1.1) AppleWebKit/413 (KHTML, like Gecko) Safari/413

```

虽然诺基亚 N 系列手机在用户代理字符串中声称使用的是"Safari",但实际上并不是 Safari,尽管确实是基于 WebKit 引擎。只要像下面检测一下用户代理字符串中是否存在"NokiaN",就足以确定是不是该系列的手机了。

```
system.nokiaN = ua.indexOf("NokiaN") > -1;
```

在了解这些设备信息的基础上,就可以通过下列代码来确定用户使用的是什么设备中的 WebKit 来访问网页:

```
if (client.engine.webkit){  
    if (client.system.iOS){  
        //iOS 手机的内容  
    } else if (client.system.android){  
        //Android 手机的内容  
    } else if (client.system.nokiaN){  
        //诺基亚手机的内容  
    }  
}
```

最后一种主要的移动设备平台是 Windows Mobile (也称为 Windows CE),用于 Pocket PC 和 Smartphone 中。由于从技术上说这些平台都属于 Windows 平台,因此 Windows 平台和操作系统都会返回正确的值。对于 Windows Mobile 5.0 及以前版本,这两种设备的用户代理字符串非常相似,如下所示:

```
Mozilla/4.0 (compatible; MSIE 4.01; Windows CE; PPC; 240x320)  
Mozilla/4.0 (compatible; MSIE 4.01; Windows CE; Smartphone; 176x220)
```

第一个来自 Pocket PC 中的移动 Internet Explorer 4.01,第二个来自 Smartphone 中的同一个浏览器。当 Windows 操作系统检测脚本中检测这两个字符串时,system.win 将被设置为"CE",因此在检测 Windows Mobile 时可以使用这个值:

```
system.winMobile = (system.win == "CE");
```

不建议测试字符串中的"PPC"或"Smartphone",因为在 Windows Mobile 5.0 以后版本的浏览器中,这些记号已经被移除了。不过,一般情况下,只知道某个设备使用的是 Windows Mobile 也就足够了。

Windows Phone 7 的用户代理字符串稍有改进,基本格式如下:

```
Mozilla/4.0 (compatible; MSIE 7.0; Windows Phone OS 7.0; Trident/3.1; IEMobile/7.0)  
Asus;Galaxy6
```

其中,Windows 操作符的标识符与以往完全不同,因此在这个用户代理中 client.system.win 等于"Ph"。从中可以取得有关系统的更多信息:

```
//windows mobile  
if (system.win == "CE"){  
    system.winMobile = system.win;  
} else if (system.win == "Ph"){  
    if(/Windows Phone OS (\d+.\d+)/.test(ua)){  
        system.win = "Phone";  
        system.winMobile = parseFloat(RegExp["$1"]);  
    }  
}
```

如果 system.win 的值是"CE",就说明是老版本的 Windows Mobile,因此 system.winMobile 会被设置为相同的值(只能知道这个信息)。如果 system.win 的值是"Ph",那么这个设备就可能是



Windows Phone 7 或更新版本。因此就用正则表达式来测试格式并提取版本号，将 system.win 的值重置为 "Phone"，而将 system.winMobile 设置为版本号。

## 6. 识别游戏系统

除了移动设备之外，视频游戏系统中的 Web 浏览器也开始日益普及。任天堂 Wii 和 Playstation 3 或者内置 Web 浏览器，或者提供了浏览器下载。Wii 中的浏览器实际上是定制版的 Opera，是专门为 Wii Remote 设计的。Playstation 的浏览器是自己开发的，没有基于前面提到的任何呈现引擎。这两个浏览器中的用户代理字符串如下所示：

```
Opera/9.10 (Nintendo Wii;U; ; 1621; en)
Mozilla/5.0 (PLAYSTATION 3; 2.00)
```

第一个字符串来自运行在 Wii 中的 Opera，它忠实地继承了 Opera 最初的用户代理字符串格式（Wii 上的 Opera 不具备隐瞒身份的能力）。第二个字符串来自 Playstation3，虽然它为了兼容性而将自己标识为 Mozilla 5.0，但并没有给出太多信息。而且，设备名称居然全部使用了大写字母，让人觉得很奇怪；强烈希望将来的版本能够改变这种情况。

在检测这些设备以前，我们必须先为 client.system 中添加适当的属性，如下所示：

```
var client = function(){

    var engine = {

        //呈现引擎
        ie: 0,
        gecko: 0,
        webkit: 0,
        khtml: 0,
        opera: 0,

        //具体的版本号
        ver: null
    };

    var browser = {

        //浏览器
        ie: 0,
        firefox: 0,
        safari: 0,
        konq: 0,
        opera: 0,
        chrome: 0,

        //具体的版本号
        ver: null
    };

    var system = {
        win: false,
        mac: false,
        xll: false,

        //移动设备
        iphone: false,
        ipod: false,
```

```

        ipad: false,
        ios: false,
        android: false,
        nokiaN: false,
        winMobile: false,
        //游戏系统
        wii: false,
        ps: false
    };

    //在此检测呈现引擎、平台和设备

    return {
        engine: engine,
        browser: browser,
        system: system
    };

}();

```

检测前述游戏系统的代码如下：

```


system.wii = ua.indexOf("Wii") > -1;
system.ps = /playstation/i.test(ua);

```

对于 Wii，只要检测字符串"Wii"就够了，而其他代码将发现这是一个 Opera 浏览器，并将正确的版本号保存在 client.browser.opera 中。对于 Playstation，我们则使用正则表达式来以不区分大小写的方式测试用户代理字符串。

### 9.3.3 完整的代码

以下是完整的用户代理字符串检测脚本，包括检测呈现引擎、平台、Windows 操作系统、移动设备和游戏系统。



```

var client = function(){
    //呈现引擎
    var engine = {
        ie: 0,
        gecko: 0,
        webkit: 0,
        khtml: 0,
        opera: 0,

        //完整的版本号
        ver: null
    };

    //浏览器
    var browser = {

        //主要浏览器
        ie: 0,
        firefox: 0,
        safari: 0,
        konq: 0,
        opera: 0,

```

```

        chrome: 0,

        //具体的版本号
        ver: null
    };

    //平台、设备和操作系统
    var system = {
        win: false,
        mac: false,
        x11: false,

        //移动设备
        iphone: false,
        ipod: false,
        ipad: false,
        ios: false,
        android: false,
        nokiaN: false,
        winMobile: false,

        //游戏系统
        wii: false,
        ps: false
    };

    //检测呈现引擎和浏览器
    var ua = navigator.userAgent;
    if (window.opera){
        engine.ver = browser.ver = window.opera.version();
        engine.opera = browser.opera = parseFloat(engine.ver);
    } else if (/AppleWebKit\/(\S+)/.test(ua)){
        engine.ver = RegExp["$1"];
        engine.webkit = parseFloat(engine.ver);

        //确定是Chrome 还是 Safari
        if (/Chrome\/(\S+)/.test(ua)){
            browser.ver = RegExp["$1"];
            browser.chrome = parseFloat(browser.ver);
        } else if (/Version\/(\S+)/.test(ua)){
            browser.ver = RegExp["$1"];
            browser.safari = parseFloat(browser.ver);
        } else {
            //近似地确定版本号
            var safariVersion = 1;
            if (engine.webkit < 100){
                safariVersion = 1;
            } else if (engine.webkit < 312){
                safariVersion = 1.2;
            } else if (engine.webkit < 412){
                safariVersion = 1.3;
            } else {
                safariVersion = 2;
            }

            browser.safari = browser.ver = safariVersion;
        }
    }

```

```

} else if (/KHTML\/(\S+)/.test(ua) || /Konqueror\/([^\;]+)/.test(ua)){
    engine.ver = browser.ver = RegExp["$1"];
    engine.khtml = browser.kong = parseFloat(engine.ver);
} else if (/rv:([^\)]+)\) Gecko\/\d{8}/.test(ua)){
    engine.ver = RegExp["$1"];
    engine.gecko = parseFloat(engine.ver);

    //确定是不是Firefox
    if (/Firefox\/(\S+)/.test(ua)){
        browser.ver = RegExp["$1"];
        browser.firefox = parseFloat(browser.ver);
    }
} else if (/MSIE ([^\;]+)/.test(ua)){
    engine.ver = browser.ver = RegExp["$1"];
    engine.ie = browser.ie = parseFloat(engine.ver);
}

//检测浏览器
browser.ie = engine.ie;
browser.opera = engine.opera;

//检测平台
var p = navigator.platform;
system.win = p.indexOf("Win") == 0;
system.mac = p.indexOf("Mac") == 0;
system.x11 = (p == "X11") || (p.indexOf("Linux") == 0);

//检测Windows 操作系统
if (system.win){
    if (/Win(?:dows)?(?:[^\d]{2})s?(\d+\.\d+)?/.test(ua)){
        if (RegExp["$1"] == "NT"){
            switch(RegExp["$2"]){
                case "5.0":
                    system.win = "2000";
                    break;
                case "5.1":
                    system.win = "XP";
                    break;
                case "6.0":
                    system.win = "Vista";
                    break;
                case "6.1":
                    system.win = "7";
                    break;
                default:
                    system.win = "NT";
                    break;
            }
        } else if (RegExp["$1"] == "9x"){
            system.win = "ME";
        } else {
            system.win = RegExp["$1"];
        }
    }
}

//移动设备

```

```

system.iphone = ua.indexOf("iPhone") > -1;
system.ipod = ua.indexOf("iPod") > -1;
system.ipad = ua.indexOf("iPad") > -1;
system.nokiaN = ua.indexOf("NokiaN") > -1;

//windows mobile
if (system.win == "CE"){
    system.winMobile = system.win;
} else if (system.win == "Ph"){
    if (/Windows Phone OS (\d+\.\d+)/.test(ua)){
        system.win = "Phone";
        system.winMobile = parseFloat(RegExp["$1"]);
    }
}

//检测 iOS 版本
if (system.mac && ua.indexOf("Mobile") > -1){
    if (/CPU (?:iPhone )?OS (\d+\.\d+)/.test(ua)){
        system.ios = parseFloat(RegExp.$1.replace("_", "."));
    } else {
        system.ios = 2; //不能真正检测出来，所以只能猜测
    }
}

//检测 Android 版本
if (/Android (\d+\.\d+)/.test(ua)){
    system.android = parseFloat(RegExp.$1);
}

//游戏系统
system.wii = ua.indexOf("Wii") > -1;
system.ps = /playstation/i.test(ua);

//返回这些对象
return {
    engine:    engine,
    browser:   browser,
    system:    system
};
}();

```

---

*client.js*

### 9.3.4 使用方法

我们在前面已经强调过了，用户代理检测是客户端检测的最后一个选择。只要可能，都应该优先采用能力检测和怪癖检测。用户代理检测一般适用于下列情形。

- ❑ 不能直接准确地使用能力检测或怪癖检测。例如，某些浏览器实现了为将来功能预留的存根（stub）函数。在这种情况下，仅测试相应的函数是否存在还得不到足够的信息。
- ❑ 同一款浏览器在不同平台下具备不同的能力。这时候，可能就有必要确定浏览器位于哪个平台下。

□ 为了跟踪分析等目的需要知道确切的浏览器。

## 9.4 小结

客户端检测是 JavaScript 开发中最具争议的一个话题。由于浏览器间存在差别，通常需要根据不同浏览器的能力分别编写不同的代码。有不少客户端检测方法，但下列是最经常使用的。

- **能力检测**：在编写代码之前先检测特定浏览器的能力。例如，脚本在调用某个函数之前，可能要先检测该函数是否存在。这种检测方法将开发人员从考虑具体的浏览器类型和版本中解放出来，让他们把注意力集中到相应的能力是否存在上。能力检测无法精确地检测特定的浏览器和版本。
- **怪癖检测**：怪癖实际上是浏览器实现中存在的 bug，例如早期的 WebKit 中就存在一个怪癖，即它会在 `for-in` 循环中返回被隐藏的属性。怪癖检测通常涉及到运行一小段代码，然后确定浏览器是否存在某个怪癖。由于怪癖检测与能力检测相比效率更低，因此应该只在某个怪癖会干扰脚本运行的情况下使用。怪癖检测无法精确地检测特定的浏览器和版本。
- **用户代理检测**：通过检测用户代理字符串来识别浏览器。用户代理字符串中包含大量与浏览器有关的信息，包括浏览器、平台、操作系统及浏览器版本。用户代理字符串有过一段相当长的发展历史，在此期间，浏览器提供商试图通过在用户代理字符串中添加一些欺骗性信息，欺骗网站相信自己的浏览器是另外一种浏览器。用户代理检测需要特殊的技巧，特别是要注意 Opera 会隐瞒其用户代理字符串的情况。即便如此，通过用户代理字符串仍然能够检测出浏览器所用的呈现引擎以及所在的平台，包括移动设备和游戏系统。

在决定使用哪种客户端检测方法时，一般应优先考虑使用能力检测。怪癖检测是确定应该如何处理代码的第二选择。而用户代理检测则是客户端检测的最后一种方案，因为这种方法对用户代理字符串具有很强的依赖性。