样可以生成便于存储和访问的结果）。

> **注意**　能力检测最适合用于决定下一步该怎么做，而不一定能够作为辨识浏览器的标志。

13.2　用户代理检测

用户代理检测通过浏览器的用户代理字符串确定使用的是什么浏览器。用户代理字符串包含在每个 HTTP 请求的头部，在 JavaScript 中可以通过 `navigator.userAgent` 访问。在服务器端，常见的做法是根据接收到的用户代理字符串确定浏览器并执行相应操作。而在客户端，用户代理检测被认为是不可靠的，只应该在没有其他选项时再考虑。

用户代理字符串最受争议的地方就是，在很长一段时间里，浏览器都通过在用户代理字符串包含错误或误导性信息来欺骗服务器。要理解背后的原因，必须回顾一下自 Web 出现之后用户代理字符串的历史。

13.2.1　用户代理的历史

HTTP 规范（1.0 和 1.1）要求浏览器应该向服务器发送包含浏览器名称和版本信息的简短字符串。RFC 2616（HTTP 1.1）是这样描述用户代理字符串的：

> 产品标记用于通过软件名称和版本来标识通信产品的身份。多数使用产品标记的字段也允许列出属于应用主要部分的子产品，以空格分隔。按照约定，产品按照标识应用重要程度的先后顺序列出。

这个规范进一步要求用户代理字符串应该是“标记/版本”形式的产品列表。但现实当中的用户代理字符串远没有那么简单。

1. 早期浏览器

美国国家超级计算应用中心（NCSA，National Center for Supercomputing Applications）发布于 1993 年的 Mosaic 是早期 Web 浏览器的代表，其用户代理字符串相当简单，类似于：

```
Mosaic/0.9
```

虽然在不同操作系统和平台中可能会有所不同，但基本形式都是这么简单直接。斜杠前是产品名称（有时候可能是“NCSA Mosaic”之类的），斜杠后是产品版本。

在网景公司准备开发浏览器时，代号确定为“Mozilla”（Mosaic Killer 的简写）。第一个公开发行版 Netscape Navigator 2 的用户代理字符串是这样的：

```
Mozilla/Version [Language] (Platform; Encryption)
```

网景公司遵守了将产品名称和版本作为用户代理字符串的规定，但又在后面添加了如下信息。

- Language：语言代码，表示浏览器的目标使用语言。
- Platform：表示浏览器所在的操作系统和/或平台。
- Encryption：包含的安全加密类型，可能的值是 U（128 位加密）、I（40 位加密）和 N（无加密）。

Netscape Navigator 2 的典型用户代理字符串如下所示：

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

这个字符串表示 Netscape Navigator 2.02，在主要使用法语地区的发行，运行在 Windows NT 上，40 位加密。总体上看，通过产品名称还是很容易知道这是什么浏览器的。

2. Netscape Navigator 3 和 IE3

1996 年，Netscape Navigator 3 发布之后超过 Mosaic 成为最受欢迎的浏览器。其用户代理字符串也发生了一些小变化，删除了语言信息，并将操作系统或系统 CPU 信息（OS-or-CPU description）等列为可选信息。此时的格式如下：

```
Mozilla/Version (Platform; Encryption [; OS-or-CPU description])
```

运行在 Windows 系统上的 Netscape Navigator 3 的典型用户代理字符串如下：

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

这个字符串表示 Netscape Navigator 3 运行在 Windows 95 上，采用了 128 位加密。注意在 Windows 系统上，没有“OS-or-CPU”部分。

Netscape Navigator 3 发布后不久，微软也首次对外发布了 IE3。这是因为当时 Netscape Navigator 是市场占有率最高的浏览器，很多服务器在返回网页之前都会特意检测其用户代理字符串。如果 IE 因此打不开网页，那么这个当时初出茅庐的浏览器就会遭受重创。为此，IE 就在用户代理字符串中添加了兼容 Netscape 用户代理字符串的内容。结果格式为：

```
Mozilla/2.0 (compatible; MSIE Version; Operating System)
```

比如，Windows 95 平台上的 IE3.02 的用户代理字符串如下：

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

当时的大多数浏览器检测程序都只看用户代理字符串中的产品名称，因此 IE 成功地将自己伪装成了 Mozilla，也就是 Netscape Navigator。这个做法引发了一些争议，因为它违反了浏览器标识的初衷。另外，真正的浏览器版本也跑到了字符串中间。

这个字符串中还有一个地方很有意思，即它将自己标识为 Mozilla 2.0 而不是 3.0。3.0 是当时市面上使用最多的版本，按理说使用这个版本更合逻辑。背后的原因至今也没有揭开，不过很可能就是当事人一时大意造成的。

3. Netscape Communicator 4 和 IE4~8

1997 年 8 月，Netscape Communicator 4 发布（这次发布将 Navigator 改成了 Communicator）。Netscape 在这个版本中仍然沿用了上一个版本的格式：

```
Mozilla/Version (Platform; Encryption [; OS-or-CPU description])
```

比如，Windows 98 上的第 4 版，其用户代理字符串就是这样的：

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

如果发布了补丁，则相应增加版本号，比如下面是 4.79 版的字符串：

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

微软在发布 IE4 时只更新了版本，格式不变：

```
Mozilla/4.0 (compatible; MSIE Version; Operating System)
```

比如，Windows 98 上运行的 IE4 的字符串如下：

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

更新版本号之后，IE 的版本号跟 Mozilla 的就一致了，识别同为第 4 代的两款浏览器也方便 了。可是，这种版本同步就此打住。在 IE4.5（只针对 Mac）面世时，Mozilla 的版本号还是 4，IE 的版本号却变了：

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

直到 IE7，Mozilla 的版本号就没有变过，比如：

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

IE8 在用户代理字符串中添加了额外的标识“Trident”，就是浏览器渲染引擎的代号。格式变成：

```
Mozilla/4.0 (compatible; MSIE Version; Operating System; Trident/TridentVersion)
```

比如：

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

这个新增的“Trident”是为了让开发者知道什么时候 IE8 运行兼容模式。在兼容模式下，MSIE 的版本会变成 7，但 Trident 的版本不变：

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

添加这个标识之后，就可以确定浏览器究竟是 IE7（没有“Trident”），还是 IE8 运行在兼容模式。

IE9 稍微升级了一下用户代理字符串的格式。Mozilla 的版本增加到了 5.0，Trident 的版本号也增加到了 5.0。IE9 的默认用户代理字符串是这样的：

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

如果 IE9 运行兼容模式，则会恢复旧版的 Mozilla 和 MSIE 版本号，但 Trident 的版本号还是 5.0。比如，下面就是 IE9 运行在 IE7 兼容模式下的字符串：

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

所有这些改变都是为了让之前的用户代理检测脚本正常运作，同时还能为新脚本提供额外的信息。

4. Gecko

Gecko 渲染引擎是 Firefox 的核心。Gecko 最初是作为通用 Mozilla 浏览器（即后来的 Netscape 6）的一部分开发的。有一个针对 Netscape 6 的用户代理字符串规范，规定了未来的版本应该如何构造这个字符串。新的格式与之前一直沿用到 4.*x* 版的格式有了很大出入：

```
Mozilla/MozillaVersion (Platform; Encryption; OS-or-CPU; Language;
  PrereleaseVersion)Gecko/GeckoVersion
  ApplicationProduct/ApplicationProductVersion
```

这个复杂的用户代理字符串包含了不少想法。下表列出了其中每一部分的含义。

字　符　串	是否必需	说　　明
MozillaVersion	是	Mozilla 版本
Platform	是	浏览器所在的平台。可能的值包括 Windows、Mac 和 X11（UNIX X-Windows）
Encryption	是	加密能力：U 表示 128 位，I 表示 40 位，N 表示无加密
OS-or-CPU	是	浏览器所在的操作系统或计算机处理器类型。如果是 Windows 平台，则这里是 Windows 的版本（如 WinNT、Win95）。如果是 Mac 平台，则这里是 CPU 类型（如 68k、PPC for PowerPC 或 MacIntel）。如果是 X11 平台，则这里是通过 uname -sm 命名得到的 UNIX 操作系统名

（续）

字 符 串	是否必需	说 明
Language	是	浏览器的目标使用语言
Prerelease Version	否	最初的设想是 Mozilla 预发布版的版本号，现在表示 Gecko 引擎的版本号
GeckoVersion	是	以 yyyymmdd 格式的日期表示的 Gecko 渲染引擎的版本
ApplicationProduct	否	使用 Gecko 的产品名称。可能是 Netscape、Firefox 等
ApplicationProductVersion	否	ApplicationProduct 的版本，区别于 MozillaVersion 和 GeckoVersion

要更好地理解 Gecko 的用户代理字符串，最好是看几个不同的基于 Gecko 的浏览器返回的字符串。

Windowx XP 上的 Netscape 6.21：

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

Linux 上的 SeaMonkey 1.1a：

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

Windows XP 上的 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 上的 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
```

所有这些字符串都表示使用的是基于 Gecko 的浏览器（只是版本不同）。有时候，相比于知道特定的浏览器，知道是不是基于 Gecko 才更重要。从第一个基于 Gecko 的浏览器发布开始，Mozilla 版本就是 5.0，一直没有变过。以后也不太可能会变。

在 Firefox 4 发布时，Mozilla 简化了用户代理字符串。主要变化包括以下几方面。

- 去掉了语言标记（即前面例子中的`"en-US"`）。
- 在浏览器使用强加密时去掉加密标记（因为是默认了）。这意味着 `I` 和 `N` 还可能出现，但 `U` 不可能出现了。
- 去掉了 Windows 平台上的平台标记，这是因为跟 OS-or-CPU 部分重复了，否则两个地方都会有 `Windows`。
- GeckoVersion 固定为`"Gecko/20100101"`。

下面是 Firefox 4 中用户代理字符串的例子：

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

5. WebKit

2003 年，苹果宣布将发布自己的浏览器 Safari。Safari 的渲染引擎叫 WebKit，是基于 Linux 平台浏览器 Konqueror 使用的渲染引擎 KHTML 开发的。几年后，WebKit 又拆分出自己的开源项目，专注于渲染引擎开发。

这个新浏览器和渲染引擎的开发者也面临与当初 IE3.0 时代同样的问题：怎样才能保证浏览器不被排除在流行的站点之外。答案就是在用户代理字符串中添加足够多的信息，让网站知道这个浏览器与其他浏览器是兼容的。于是 Safari 就有了下面这样的用户代理字符串：

```
Mozilla/5.0 (Platform; Encryption; OS-or-CPU; Language)
  AppleWebKit/AppleWebKitVersion (KHTML, like Gecko) Safari/SafariVersion
```

下面是一个实际的例子：

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

这个字符串也很长，不仅包括苹果 WebKit 的版本，也包含 Safari 的版本。一开始还有是否需要将浏览器标识为 Mozilla 的争论，但考虑到兼容性很快就达成了一致。现在，所有基于 WebKit 的浏览器都将自己标识为 Mozilla 5.0，与所有基于 Gecko 的浏览器一样。Safari 版本通常是浏览器的构建编号，不一定表示发布的版本号。比如 Safari 1.25 在用户代理字符串中的版本是 125.1，但也不一定始终这样对应。

Safari 用户代理字符串中最受争议的部分是在 1.0 预发布版中添加的"(KHTML, like Gecko)"。由于有意想让客户端和服务器把 Safari 当成基于 Gecko 的浏览器（好像光添加"Mozilla/5.0"还不够），苹果也招来了很多开发者的反对。苹果的回应与微软当初 IE 遭受质疑时一样：Safari 与 Mozilla 兼容，不能让网站以为用户使用了不受支持的浏览器而把 Safari 排斥在外。

Safari 的用户代理字符串在第 3 版时有所改进。下面的版本标记现在用来表示 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
```

注意这个变化只针对 Safari 而不包括 WebKit。因此，其他基于 WebKit 的浏览器可能不会有这个变化。一般来说，与 Gecko 一样，通常识别是不是 WebKit 比识别是不是 Safari 更重要。

6. Konqueror

Konqueror 是与 KDE Linux 桌面环境打包发布的浏览器，基于开源渲染引擎 KHTML。虽然只有 Linux 平台的版本，Konqueror 的用户却不少。为实现最大化兼容，Konqueror 决定采用 Internet Explore 的用户代理字符串格式：

```
Mozilla/5.0 (compatible; Konqueror/Version; OS-or-CPU)
```

不过，Konqueror 3.2 为了与 WebKit 就标识为 KHTML 保持一致，也对格式做了一点修改：

```
Mozilla/5.0 (compatible; Konqueror/Version; OS-or-CPU) KHTML/KHTMLVersion
  (like Gecko)
```

下面是一个例子：

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

Konqueror 和 KHTML 的版本号通常是一致的，有时候也只有子版本号不同。比如 Konqueror 是 3.5，而 KHTML 是 3.5.1。

7. Chrome

谷歌的 Chrome 浏览器使用 Blink 作为渲染引擎，使用 V8 作为 JavaScript 引擎。Chrome 的用户代理字符串包含所有 WebKit 的信息，另外又加上了 Chrome 及其版本的信息。其格式如下所示：

```
Mozilla/5.0 (Platform; Encryption; OS-or-CPU; Language)
  AppleWebKit/AppleWebKitVersion (KHTML, like Gecko)
  Chrome/ChromeVersion Safari/SafariVersion
```

以下是 Chrome 7 完整的用户代理字符串：

```
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
```

其中的 Safari 版本和 WebKit 版本有可能始终保持一致，但也不能肯定。

8. Opera

在用户代理字符串方面引发争议最大的一个浏览器就是 Opera。Opera 默认的用户代理字符串是所有现代浏览器中最符合逻辑的，因为它正确标识了自己和版本。在 Opera 8 之前，其用户代理字符串都是这个格式：

```
Opera/Version (OS-or-CPU; Encryption) [Language]
```

比如，Windows XP 上的 Opera 7.54 的字符串是这样的：

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

Opera 8 发布后，语言标记从括号外挪到了括号内，目的是与其他浏览器保持一致：

```
Opera/Version (OS-or-CPU; Encryption; Language)
```

Windows XP 上的 Opera 8 的字符串是这样的：

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

默认情况下，Opera 会返回这个简单的用户代理字符串。这是唯一一个使用产品名称和版本完全标识自身的主流浏览器。不过，与其他浏览器一样，Opera 也遇到了使用这种字符串的问题。虽然从技术角度看这是正确的，但网上已经有了很多浏览器检测代码只考虑 Mozilla 这个产品名称。还有不少代码专门针对 IE 或 Gecko。为了不让这些检测代码判断错误，Opera 坚持使用唯一标识自身的字符串。

从 Opera 9 开始，Opera 也采用了两个策略改变自己的字符串。一是把自己标识为别的浏览器，如 Firefox 或 IE。这时候的字符串跟 Firefox 和 IE 的一样，只不过末尾会多一个"Opera"及其版本号。比如：

```
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
```

第一个字符串把 Opera 9.5 标识为 Firefox 2，同时保持了 Opera 版本信息。第二个字符串把 Opera 9.5 标识为 IE6，也保持了 Opera 版本信息。虽然这些字符串可以通过针对 Firefox 和 IE 的测试，但也可以被识别为 Opera。

另一个策略是伪装成 Firefox 或 IE。这种情况下的用户代理字符串与 Firefox 和 IE 返回的一样，末尾也没有"Opera"及其版本信息。这样就根本没办法区分 Opera 与其他浏览器了。更严重的是，Opera 还会根据访问的网站不同设置不同的用户代理字符串，却不通知用户。比如，导航到 My Yahoo 网站会导致 Opera 将自己伪装成 Firefox。这就导致很难通过用户代理字符串来识别 Opera。

> **注意** 在 Opera 7 之前的版本中，Opera 可以解析 Windows 操作系统字符串的含义。比如，Windows NT 5.1 实际上表示 Windows XP。因此 Opera 6 的用户代理字符串中会包含 Windows XP 而不是 Windows NT 5.1。为了与其他浏览器表现更一致，Opera 7 及后来的版本就改为使用官方报告的操作系统字符串，而不是自己转换的了。

Opera 10 又修改了字符串格式，变成了下面这样：

```
Opera/9.80 (OS-or-CPU; Encryption; Language) Presto/PrestoVersion Version/Version
```

注意开头的版本号 Opera/9.80 是固定不变的。Opera 没有 9.8 这个版本，但 Opera 工程师担心某些浏览器检测脚本会错误地把 Opera/10.0 当成 Opera 1 而不是 Opera 10。因此，Opera 10 新增了额外的

Presto 标识（Presto 是 Opera 的渲染引擎）和版本标识。比如，下面是 Windows 7 上的 Opera 10.63 的字符串：

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

Opera 最近的版本已经改为在更标准的字符串末尾追加"OPR"标识符和版本号。这样，除了末尾的"OPR"标识符和版本号，字符串的其他部分与 WebKit 浏览器是类似的。下面就是 Windows 10 上的 Opera 52 的用户代理字符串：

```
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/65.0.3325.181 Safari/537.36 OPR/52.0.2871.64
```

9. iOS 与 Android

iOS 和 Android 移动操作系统上默认的浏览器都是基于 WebKit 的，因此具有与相应桌面浏览器一样的用户代理字符串。iOS 设备遵循以下基本格式：

```
Mozilla/5.0 (Platform; Encryption; OS-or-CPU like Mac OS X; Language)
  AppleWebKit/AppleWebKitVersion (KHTML, like Gecko) Version/BrowserVersion
  Mobile/MobileVersion Safari/SafariVersion
```

注意其中用于辅助判断 Mac 操作系统的"like Mac OS X"和"Mobile"相关的标识。这里的 Mobile 标识除了说明这是移动 WebKit 之外并没有什么用。平台可能是"iPhone"、"iPod"或"iPad"，因设备而异。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
```

注意在 iOS 3 以前，操作系统的版本号不会出现在用户代理字符串中。

默认的 Android 浏览器通常与 iOS 上的浏览器格式相同，只是没有 Mobile 后面的版本号（"Mobile"标识还有）。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
```

这个用户代理字符串是谷歌 Nexus One 手机上的默认浏览器的。不过，其他 Android 设备上的浏览器也遵循相同的模式。

13.2.2　浏览器分析

想要知道自己代码运行在什么浏览器上，大部分开发者会分析 window.navigator.userAgent 返回的字符串值。所有浏览器都会提供这个值，如果相信这些返回值并基于给定的一组浏览器检测这个字符串，最终会得到关于浏览器和操作系统的比较精确的结果。

相比于能力检测，用户代理检测还是有一定优势的。能力检测可以保证脚本不必理会浏览器而正常执行。现代浏览器用户代理字符串的过去、现在和未来格式都是有章可循的，我们能够利用它们准确识别浏览器。

1. 伪造用户代理

通过检测用户代理来识别浏览器并不是完美的方式，毕竟这个字符串是可以造假的。只不过实现 window.navigator 对象的浏览器（即所有现代浏览器）都会提供 userAgent 这个只读属性。因此，简单地给这个属性设置其他值不会有效：

```
console.log(window.navigator.userAgent);
// Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/65.0.3325.181 Safari/537.36

window.navigator.userAgent = 'foobar';

console.log(window.navigator.userAgent);
// Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/65.0.3325.181 Safari/537.36
```

不过，通过简单的办法可以绕过这个限制。比如，有些浏览器提供伪私有的__defineGetter__方法，利用它可以篡改用户代理字符串：

```
console.log(window.navigator.userAgent);
// Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/65.0.3325.181 Safari/537.36

window.navigator.__defineGetter__('userAgent', () => 'foobar');

console.log(window.navigator.userAgent);
// foobar
```

对付这种造假是一件吃力不讨好的事。检测用户代理是否以这种方式被篡改过是可能的，但总体来看还是一场猫捉老鼠的游戏。

与其劳心费力检测造假，不如更好地专注于浏览器识别。如果相信浏览器返回的用户代理字符串，那就可以用它来判断浏览器。如果怀疑脚本或浏览器可能篡改这个值，那最好还是使用能力检测。

2. 分析浏览器

通过解析浏览器返回的用户代理字符串，可以极其准确地推断出下列相关的环境信息：

- 浏览器
- 浏览器版本
- 浏览器渲染引擎
- 设备类型（桌面/移动）
- 设备生产商
- 设备型号
- 操作系统
- 操作系统版本

当然，新浏览器、新操作系统和新硬件设备随时可能出现，其中很多可能有着类似但并不相同的用户代理字符串。因此，用户代理解析程序需要与时俱进，频繁更新，以免落伍。自己手写的解析程序如果不及时更新或修订，很容易就过时了。本书上一版写过一个用户代理解析程序，但这一版并不推荐读者自己从头再写一个。相反，这里推荐一些 GitHub 上维护比较频繁的第三方用户代理解析程序：

- Bowser
- UAParser.js
- Platform.js
- CURRENT-DEVICE
- Google Closure
- Mootools

> **注意**　Mozilla 维基有一个页面“Compatibility/UADetectionLibraries”，其中提供了用户代理解析程序的列表，可以用来识别 Mozilla 浏览器（甚至所有主流浏览器）。这些解析程序是按照语言分组的。这个页面好像维护不频繁，但其中给出了所有主流的解析库。（注意 JavaScript 部分包含客户端库和 Node.js 库。）GitHub 上的文章“Are We Detectable Yet?”中还有一张可视化的表格，能让我们对这些库的检测能力一目了然。

13.3　软件与硬件检测

现代浏览器提供了一组与页面执行环境相关的信息，包括浏览器、操作系统、硬件和周边设备信息。这些属性可以通过暴露在 `window.navigator` 上的一组 API 获得。不过，这些 API 的跨浏览器支持还不够好，远未达到标准化的程度。

> **注意**　强烈建议在使用这些 API 之前先检测它们是否存在，因为其中多数都不是强制性的，且很多浏览器没有支持。另外，本节介绍的特性有时候不一定可靠。

13.3.1　识别浏览器与操作系统

特性检测和用户代理字符串解析是当前常用的两种识别浏览器的方式。而 `navigator` 和 `screen` 对象也提供了关于页面所在软件环境的信息。

1. **navigator.oscpu**

`navigator.oscpu` 属性是一个字符串，通常对应用户代理字符串中操作系统/系统架构相关信息。根据 HTML 实时标准：

> `oscpu` 属性的获取方法必须返回空字符串或者表示浏览器所在平台的字符串，比如`"Windows NT 10.0; Win64; x64"`或`"Linux x86_64"`。

比如，Windows 10 上的 Firefox 的 `oscpu` 属性应该对应于以下加粗的部分：

```
console.log(navigator.userAgent);
"Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:58.0) Gecko/20100101 Firefox/58.0"
console.log(navigator.oscpu);
"Windows NT 10.0; Win64; x64"
```

2. **navigator.vendor**

`navigator.vendor` 属性是一个字符串，通常包含浏览器开发商信息。返回这个字符串是浏览器 `navigator` 兼容模式的一个功能。根据 HTML 实时标准：

> `navigator.vendor` 返回一个空字符串，也可能返回字符串`"Apple Computer, Inc."`或字符串`"Google Inc."`。

例如，Chrome 中的这个 `navigator.vendor` 属性返回下面的字符串：

```
console.log(navigator.vendor); // "Google Inc."
```

3. **navigator.platform**

`navigator.platform` 属性是一个字符串，通常表示浏览器所在的操作系统。根据 HTML 实时标准：

navigator.platform 必须返回一个字符串或表示浏览器所在平台的字符串，例如"MacIntel"、"Win32"、"FreeBSD i386"或"WebTV OS"。

例如，Windows 系统下 Chrome 中的这个 navigator.platform 属性返回下面的字符串：

```
console.log(navigator.platform); // "Win32"
```

4. screen.colorDepth 和 screen.pixelDepth

screen.colorDepth 和 screen.pixelDepth 返回一样的值，即显示器每像素颜色的位深。根据 CSS 对象模型（CSSOM）规范：

screen.colorDepth 和 screen.pixelDepth 属性应该返回输出设备中每像素用于显示颜色的位数，不包含 alpha 通道。

Chrome 中这两个属性的值如下所示：

```
console.log(screen.colorDepth); // 24
console.log(screen.pixelDepth); // 24
```

5. screen.orientation

screen.orientation 属性返回一个 ScreenOrientation 对象，其中包含 Screen Orientation API 定义的屏幕信息。这里面最有意思的属性是 angle 和 type，前者返回相对于默认状态下屏幕的角度，后者返回以下 4 种枚举值之一：

- portrait-primary
- portrait-secondary
- landscape-primary
- landscape-secondary

例如，在 Chrome 移动版中，screen.orientation 返回的信息如下：

```
// 垂直看
console.log(screen.orientation.type);   // portrait-primary
console.log(screen.orientation.angle);  // 0

// 向左转
console.log(screen.orientation.type);   // landscape-primary
console.log(screen.orientation.angle);  // 90

// 向右转
console.log(screen.orientation.type);   // landscape-secondary
console.log(screen.orientation.angle);  // 270
```

根据规范，这些值的初始化取决于浏览器和设备状态。因此，不能假设 portrait-primary 和 0 始终是初始值。这两个值主要用于确定设备旋转后浏览器的朝向变化。

13.3.2 浏览器元数据

navigator 对象暴露出一些 API，可以提供浏览器和操作系统的状态信息。

1. Geolocation API

navigator.geolocation 属性暴露了 Geolocation API，可以让浏览器脚本感知当前设备的地理位置。这个 API 只在安全执行环境（通过 HTTPS 获取的脚本）中可用。

这个 API 可以查询宿主系统并尽可能精确地返回设备的位置信息。根据宿主系统的硬件和配置，返