（续）

属性/方法	说　明
locks	返回暴露 Web Locks API 的 LockManager 对象
mediaCapabilities	返回暴露 Media Capabilities API 的 MediaCapabilities 对象
mediaDevices	返回可用的媒体设备
maxTouchPoints	返回设备触摸屏支持的最大触点数
mimeTypes	返回浏览器中注册的 MIME 类型数组
onLine	返回布尔值，表示浏览器是否联网
oscpu	返回浏览器运行设备的操作系统和（或）CPU
permissions	返回暴露 Permissions API 的 Permissions 对象
platform	返回浏览器运行的系统平台
plugins	返回浏览器安装的插件数组。在 IE 中，这个数组包含页面中所有<embed>元素
product	返回产品名称（通常是"Gecko"）
productSub	返回产品的额外信息（通常是 Gecko 的版本）
registerProtocolHandler()	将一个网站注册为特定协议的处理程序
requestMediaKeySystemAccess()	返回一个期约，解决为 MediaKeySystemAccess 对象
sendBeacon()	异步传输一些小数据
serviceWorker	返回用来与 ServiceWorker 实例交互的 ServiceWorkerContainer
share()	返回当前平台的原生共享机制
storage	返回暴露 Storage API 的 StorageManager 对象
userAgent	返回浏览器的用户代理字符串
vendor	返回浏览器的厂商名称
vendorSub	返回浏览器厂商的更多信息
vibrate()	触发设备振动
webdriver	返回浏览器当前是否被自动化程序控制

navigator 对象的属性通常用于确定浏览器的类型。

12.3.1　检测插件

检测浏览器是否安装了某个插件是开发中常见的需求。除 IE10 及更低版本外的浏览器，都可以通过 plugins 数组来确定。这个数组中的每一项都包含如下属性。

- name：插件名称。
- description：插件介绍。
- filename：插件的文件名。
- length：由当前插件处理的 MIME 类型数量。

通常，name 属性包含识别插件所需的必要信息，尽管不是特别准确。检测插件就是遍历浏览器中可用的插件，并逐个比较插件的名称，如下所示：

```
// 插件检测，IE10 及更低版本无效
let hasPlugin = function(name) {
  name = name.toLowerCase();
```

```
  for (let plugin of window.navigator.plugins){
    if (plugin.name.toLowerCase().indexOf(name) > -1){
      return true;
    }
  }

  return false;
}

// 检测 Flash
alert(hasPlugin("Flash"));

// 检测 QuickTime
alert(hasPlugin("QuickTime"));
```

这个 hasPlugin()方法接收一个参数，即待检测插件的名称。第一步是把插件名称转换为小写形式，以便于比较。然后，遍历 plugins 数组，通过 indexOf()方法检测每个 name 属性，看传入的名称是不是存在于某个数组中。比较的字符串全部小写，可以避免大小写问题。传入的参数应该尽可能独一无二，以避免混淆。像"Flash"、"QuickTime"这样的字符串就可以避免混淆。这个方法可以在 Firefox、Safari、Opera 和 Chrome 中检测插件。

> **注意** plugins 数组中的每个插件对象还有一个 MimeType 对象，可以通过中括号访问。每个 MimeType 对象有 4 个属性：description 描述 MIME 类型，enabledPlugin 是指向插件对象的指针，suffixes 是该 MIME 类型对应扩展名的逗号分隔的字符串，type 是完整的 MIME 类型字符串。

IE11 的 window.navigator 对象开始支持 plugins 和 mimeTypes 属性。这意味着前面定义的函数可以适用于所有较新版本的浏览器。而且，IE11 中的 ActiveXObject 也从 DOM 中隐身了，意味着不能再用它来作为检测特性的手段。

旧版本 IE 中的插件检测

IE10 及更低版本中检测插件的问题比较多，因为这些浏览器不支持 Netscape 式的插件。在这些 IE 中检测插件要使用专有的 ActiveXObject，并尝试实例化特定的插件。IE 中的插件是实现为 COM 对象的，由唯一的字符串标识。因此，要检测某个插件就必须知道其 COM 标识符。例如，Flash 的标识符是"ShockwaveFlash.ShockwaveFlash"。知道了这个信息后，就可以像这样检测 IE 中是否安装了 Flash：

```
// 在旧版本 IE 中检测插件
function hasIEPlugin(name) {
  try {
    new ActiveXObject(name);
    return true;
  } catch (ex) {
    return false;
  }
}

// 检测 Flash
alert(hasIEPlugin("ShockwaveFlash.ShockwaveFlash"));

// 检测 QuickTime
alert(hasIEPlugin("QuickTime.QuickTime"));
```

在这个例子中，hasIEPlugin()函数接收一个 DOM 标识符参数。为检测插件，这个函数会使用传入的标识符创建一个新 ActiveXObject 实例。相应代码封装在一个 try/catch 语句中，因此如果创建的插件不存在则会抛出错误。如果创建成功则返回 true，如果失败则在 catch 块中返回 false。上面的例子还演示了如何检测 Flash 和 QuickTime 插件。

因为检测插件涉及两种方式，所以一般要针对特定插件写一个函数，而不是使用通常的检测函数。比如下面的例子：

```
// 在所有浏览器中检测 Flash
function hasFlash() {
  var result = hasPlugin("Flash");
  if (!result){
    result = hasIEPlugin("ShockwaveFlash.ShockwaveFlash");
  }
  return result;
}

// 在所有浏览器中检测 QuickTime
function hasQuickTime() {
  var result = hasPlugin("QuickTime");
  if (!result){
    result = hasIEPlugin("QuickTime.QuickTime");
  }
  return result;
}

// 检测 Flash
alert(hasFlash());

// 检测 QuickTime
alert(hasQuickTime());
```

以上代码定义了两个函数 hasFlash()和 hasQuickTime()。每个函数都先尝试使用非 IE 插件检测方式，如果返回 false（对 IE 可能会），则再使用 IE 插件检测方式。如果 IE 插件检测方式再返回 false，整个检测方法也返回 false。只要有一种方式返回 true，检测方法就会返回 true。

> **注意**　plugins 有一个 refresh()方法，用于刷新 plugins 属性以反映新安装的插件。这个方法接收一个布尔值参数，表示刷新时是否重新加载页面。如果传入 true，则所有包含插件的页面都会重新加载。否则，只有 plugins 会更新，但页面不会重新加载。

12.3.2　注册处理程序

现代浏览器支持 navigator 上的（在 HTML5 中定义的）registerProtocolHandler()方法。这个方法可以把一个网站注册为处理某种特定类型信息应用程序。随着在线 RSS 阅读器和电子邮件客户端的流行，可以借助这个方法将 Web 应用程序注册为像桌面软件一样的默认应用程序。

要使用 registerProtocolHandler()方法，必须传入 3 个参数：要处理的协议（如"mailto"或"ftp"）、处理该协议的 URL，以及应用名称。比如，要把一个 Web 应用程序注册为默认邮件客户端，可以这样做：

```
navigator.registerProtocolHandler("mailto",
  "http://www.somemailclient.com?cmd=%s",
  "Some Mail Client");
```

这个例子为"mailto"协议注册了一个处理程序，这样邮件地址就可以通过指定的 Web 应用程序打开。注意，第二个参数是负责处理请求的 URL，%s 表示原始的请求。

12.4 screen 对象

window 的另一个属性 screen 对象，是为数不多的几个在编程中很少用的 JavaScript 对象。这个对象中保存的纯粹是客户端能力信息，也就是浏览器窗口外面的客户端显示器的信息，比如像素宽度和像素高度。每个浏览器都会在 screen 对象上暴露不同的属性。下表总结了这些属性。

属　性	说　明
availHeight	屏幕像素高度减去系统组件高度（只读）
availLeft	没有被系统组件占用的屏幕的最左侧像素（只读）
availTop	没有被系统组件占用的屏幕的最顶端像素（只读）
availWidth	屏幕像素宽度减去系统组件宽度（只读）
colorDepth	表示屏幕颜色的位数；多数系统是 32（只读）
height	屏幕像素高度
left	当前屏幕左边的像素距离
pixelDepth	屏幕的位深（只读）
top	当前屏幕顶端的像素距离
width	屏幕像素宽度
orientation	返回 Screen Orientation API 中屏幕的朝向

12.5 history 对象

history 对象表示当前窗口首次使用以来用户的导航历史记录。因为 history 是 window 的属性，所以每个 window 都有自己的 history 对象。出于安全考虑，这个对象不会暴露用户访问过的 URL，但可以通过它在不知道实际 URL 的情况下前进和后退。

12.5.1 导航

go()方法可以在用户历史记录中沿任何方向导航，可以前进也可以后退。这个方法只接收一个参数，这个参数可以是一个整数，表示前进或后退多少步。负值表示在历史记录中后退（类似点击浏览器的“后退”按钮），而正值表示在历史记录中前进（类似点击浏览器的“前进”按钮）。下面来看几个例子：

```
// 后退一页
history.go(-1);

// 前进一页
history.go(1);

// 前进两页
history.go(2);
```

在旧版本的一些浏览器中，go()方法的参数也可以是一个字符串，这种情况下浏览器会导航到历史中包含该字符串的第一个位置。最接近的位置可能涉及后退，也可能涉及前进。如果历史记录中没有匹配的项，则这个方法什么也不做，如下所示：

```
// 导航到最近的 wrox.com 页面
history.go("wrox.com");

// 导航到最近的 nczonline.net 页面
history.go("nczonline.net");
```

go()有两个简写方法：back()和 forward()。顾名思义，这两个方法模拟了浏览器的后退按钮和前进按钮：

```
// 后退一页
history.back();

// 前进一页
history.forward();
```

history 对象还有一个 length 属性，表示历史记录中有多个条目。这个属性反映了历史记录的数量，包括可以前进和后退的页面。对于窗口或标签页中加载的第一个页面，history.length 等于 1。通过以下方法测试这个值，可以确定用户浏览器的起点是不是你的页面：

```
if (history.length == 1){
  // 这是用户窗口中的第一个页面
}
```

history 对象通常被用于创建“后退”和“前进”按钮，以及确定页面是不是用户历史记录中的第一条记录。

> **注意**　如果页面 URL 发生变化，则会在历史记录中生成一个新条目。对于 2009 年以来发布的主流浏览器，这包括改变 URL 的散列值（因此，把 location.hash 设置为一个新值会在这些浏览器的历史记录中增加一条记录）。这个行为常被单页应用程序框架用来模拟前进和后退，这样做是为了不会因导航而触发页面刷新。

12.5.2　历史状态管理

现代 Web 应用程序开发中最难的环节之一就是历史记录管理。用户每次点击都会触发页面刷新的时代早已过去，“后退”和“前进”按钮对用户来说就代表“帮我切换一个状态”的历史也就随之结束了。为解决这个问题，首先出现的是 hashchange 事件（第 17 章介绍事件时会讨论）。HTML5 也为 history 对象增加了方便的状态管理特性。

hashchange 会在页面 URL 的散列变化时被触发，开发者可以在此时执行某些操作。而状态管理 API 则可以让开发者改变浏览器 URL 而不会加载新页面。为此，可以使用 history.pushState()方法。这个方法接收 3 个参数：一个 state 对象、一个新状态的标题和一个（可选的）相对 URL。例如：

```
let stateObject = {foo:"bar"};

history.pushState(stateObject, "My title", "baz.html");
```

pushState()方法执行后，状态信息就会被推到历史记录中，浏览器地址栏也会改变以反映新的相对 URL。除了这些变化之外，即使 location.href 返回的是地址栏中的内容，浏览器页不会向服务器

发送请求。第二个参数并未被当前实现所使用，因此既可以传一个空字符串也可以传一个短标题。第一个参数应该包含正确初始化页面状态所必需的信息。为防止滥用，这个状态的对象大小是有限制的，通常在 500KB ~ 1MB 以内。

因为 pushState()会创建新的历史记录，所以也会相应地启用“后退”按钮。此时单击“后退”按钮，就会触发 window 对象上的 popstate 事件。popstate 事件的事件对象有一个 state 属性，其中包含通过 pushState()第一个参数传入的 state 对象：

```
window.addEventListener("popstate", (event) => {
  let state = event.state;
  if (state) { // 第一个页面加载时状态是 null
    processState(state);
  }
});
```

基于这个状态，应该把页面重置为状态对象所表示的状态（因为浏览器不会自动为你做这些）。记住，页面初次加载时没有状态。因此点击“后退”按钮直到返回最初页面时，event.state 会为 null。

可以通过 history.state 获取当前的状态对象，也可以使用 replaceState()并传入与 pushState()同样的前两个参数来更新状态。更新状态不会创建新历史记录，只会覆盖当前状态：

```
history.replaceState({newFoo: "newBar"}, "New title");
```

传给 pushState()和 replaceState()的 state 对象应该只包含可以被序列化的信息。因此，DOM 元素之类并不适合放到状态对象里保存。

> **注意** 使用 HTML5 状态管理时，要确保通过 pushState()创建的每个“假”URL 背后都对应着服务器上一个真实的物理 URL。否则，单击“刷新”按钮会导致 404 错误。所有单页应用程序（SPA，Single Page Application）框架都必须通过服务器或客户端的某些配置解决这个问题。

12.6 小结

浏览器对象模型（BOM，Browser Object Model）是以 window 对象为基础的，这个对象代表了浏览器窗口和页面可见的区域。window 对象也被复用为 ECMAScript 的 Global 对象，因此所有全局变量和函数都是它的属性，而且所有原生类型的构造函数和普通函数也都从一开始就存在于这个对象之上。本章讨论了 BOM 的以下内容。

- 要引用其他 window 对象，可以使用几个不同的窗口指针。
- 通过 location 对象可以以编程方式操纵浏览器的导航系统。通过设置这个对象上的属性，可以改变浏览器 URL 中的某一部分或全部。
- 使用 replace()方法可以替换浏览器历史记录中当前显示的页面，并导航到新 URL。
- navigator 对象提供关于浏览器的信息。提供的信息类型取决于浏览器，不过有些属性如 userAgent 是所有浏览器都支持的。

BOM 中的另外两个对象也提供了一些功能。screen 对象中保存着客户端显示器的信息。这些信息通常用于评估浏览网站的设备信息。history 对象提供了操纵浏览器历史记录的能力，开发者可以确定历史记录中包含多少个条目，并以编程方式实现在历史记录中导航，而且也可以修改历史记录。

第13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
- 软件与硬件检测
- 检测策略

虽然浏览器厂商齐心协力想要实现一致的接口，但事实上仍然是每家浏览器都有自己的长处与不足。跨平台的浏览器尽管版本相同，但总会存在不同的问题。这些差异迫使 Web 开发者要么面向最大公约数而设计，要么（更常见地）使用各种方法来检测客户端，以克服或避免这些缺陷。

客户端检测一直是 Web 开发中饱受争议的话题，这些话题普遍围绕所有浏览器应支持一系列公共特性，理想情况下是这样的。而现实当中，浏览器之间的差异和莫名其妙的行为，让客户端检测变成一种补救措施，而且也成为了开发策略的重要一环。如今，浏览器之间的差异相对 IE 大溃败以前已经好很多了，但浏览器间的不一致性依旧是 Web 开发中的常见主题。

要检测当前的浏览器有很多方法，每一种都有各自的长处和不足。问题的关键在于知道客户端检测应该是解决问题的最后一个举措。任何时候，只要有更普适的方案可选，都应该毫不犹豫地选择。首先要设计最常用的方案，然后再考虑为特定的浏览器进行补救。

13.1 能力检测

能力检测（又称特性检测）即在 JavaScript 运行时中使用一套简单的检测逻辑，测试浏览器是否支持某种特性。这种方式不要求事先知道特定浏览器的信息，只需检测自己关心的能力是否存在即可。能力检测的基本模式如下：

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

比如，IE5 之前的版本中没有 `document.getElementById()` 这个 DOM 方法，但可以通过 `document.all` 属性实现同样的功能。为此，可以进行如下能力检测：

```
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!");
  }
}
```

这个 getElement()函数的目的是根据给定的 ID 获取元素。因为标准的方式是使用 document.getElementById()，所以首先测试它。如果这个函数存在（不是 undefined），那就使用这个方法；否则检测 document.all 是否存在，如果存在则使用。如果这两个能力都不存在（基本上不可能），则抛出错误说明功能无法实现。

能力检测的关键是理解两个重要概念。首先，如前所述，应该先检测最常用的方式。在前面的例子中就是先检测 document.getElementById()再检测 document.all。测试最常用的方案可以优化代码执行，这是因为在多数情况下都可以避免无谓检测。

其次是必须检测切实需要的特性。某个能力存在并不代表别的能力也存在。比如下面的例子：

```
function getWindowWidth() {
  if (document.all) { // 假设 IE
    return document.documentElement.clientWidth; // 不正确的用法!
  } else {
    return window.innerWidth;
  }
}
```

这个例子展示了不正确的能力检测方式。getWindowWidth()函数首先检测 document.all 是否存在，如果存在则返回 document.documentElement.clientWidth，理由是 IE8 及更低版本不支持 window.innerWidth。这个例子的问题在于检测到 document.all 存在并不意味着浏览器是 IE。事实，也可能是某个早期版本的 Opera，既支持 document.all 也支持 windown.innerWidth。

13.1.1 安全能力检测

能力检测最有效的场景是检测能力是否存在的同时，验证其是否能够展现出预期的行为。前一节中的例子依赖将测试对象的成员转换类型，然后再确定它是否存在。虽然这样能够确定检测的对象成员存在，但不能确定它就是你想要的。来看下面的例子，这个函数尝试检测某个对象是否可以排序：

```
// 不要这样做! 错误的能力检测，只能检测到能力是否存在
function isSortable(object) {
  return !!object.sort;
}
```

这个函数尝试通过检测对象上是否有 sort()方法来确定它是否支持排序。问题在于，即使这个对象有一个 sort 属性，这个函数也会返回 true：

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

简单地测试到一个属性存在并不代表这个对象就可以排序。更好的方式是检测 sort 是不是函数：

```
// 好一些，检测 sort 是不是函数
function isSortable(object) {
  return typeof object.sort == "function";
}
```

上面的代码中使用的 typeof 操作符可以确定 sort 是不是函数，从而确认是否可以调用它对数据进行排序。

进行能力检测时应该尽量使用 typeof 操作符，但光有它还不够。尤其是某些宿主对象并不保证对 typeof 测试返回合理的值。最有名的例子就是 Internet Explorer（IE）。在多数浏览器中，下面的代码都会在 document.createElement()存在时返回 true：

```
// 不适用于 IE8 及更低版本
function hasCreateElement() {
  return typeof document.createElement == "function";
}
```

但在 IE8 及更低版本中，这个函数会返回 false。这是因为 typeof document.createElement 返回"object"而非"function"。前面提到过，DOM 对象是宿主对象，而宿主对象在 IE8 及更低版本中是通过 COM 而非 JScript 实现的。因此，document.createElement()函数被实现为 COM 对象，typeof 返回"object"。IE9 对 DOM 方法会返回"function"。

> **注意**　要深入了解 JavaScript 能力检测，推荐阅读 Peter Michaux 的文章“Feature Detection—State of the Art Browser Scripting”。

13.1.2　基于能力检测进行浏览器分析

虽然可能有人觉得能力检测类似于黑科技，但恰当地使用能力检测可以精准地分析运行代码的浏览器。使用能力检测而非用户代理检测的优点在于，伪造用户代理字符串很简单，而伪造能够欺骗能力检测的浏览器特性却很难。

1. 检测特性

可以按照能力将浏览器归类。如果你的应用程序需要使用特定的浏览器能力，那么最好集中检测所有能力，而不是等到用的时候再重复检测。比如：

```
// 检测浏览器是否支持 Netscape 式的插件
let hasNSPlugins = !!(navigator.plugins && navigator.plugins.length);

// 检测浏览器是否具有 DOM Level 1 能力
let hasDOM1 = !!(document.getElementById && document.createElement &&
         document.getElementsByTagName);
```

这个例子完成了两项检测：一项是确定浏览器是否支持 Netscape 式的插件，另一项是检测浏览器是否具有 DOM Level 1 能力。保存在变量中的布尔值可以用在后面的条件语句中，这样比重复检测省事多了。

2. 检测浏览器

可以根据对浏览器特性的检测并与已知特性对比，确认用户使用的是什么浏览器。这样可以获得比用户代码嗅探（稍后讨论）更准确的结果。但未来的浏览器版本可能不适用于这套方案。

下面来看一个例子，根据不同浏览器独有的行为推断出浏览器的身份。这里故意没有使用 navigator.userAgent 属性，后面会讨论它：

```
class BrowserDetector {
  constructor() {
    // 测试条件编译
    // IE6~10 支持
    this.isIE_Gte6Lte10 = /*@cc_on!@*/false;

    // 测试 documentMode
    // IE7~11 支持
    this.isIE_Gte7Lte11 = !!document.documentMode;
```

```
        // 测试 StyleMedia 构造函数
        // Edge 20 及以上版本支持
        this.isEdge_Gte20 = !!window.StyleMedia;

        // 测试 Firefox 专有扩展安装 API
        // 所有版本的 Firefox 都支持
        this.isFirefox_Gte1 = typeof InstallTrigger !== 'undefined';

        // 测试 chrome 对象及其 webstore 属性
        // Opera 的某些版本有 window.chrome，但没有 window.chrome.webstore
        // 所有版本的 Chrome 都支持
        this.isChrome_Gte1 = !!window.chrome && !!window.chrome.webstore;

        // Safari 早期版本会给构造函数的标签符追加"Constructor"字样，如：
        // window.Element.toString(); // [object ElementConstructor]
        // Safari 3~9.1 支持
        this.isSafari_Gte3Lte9_1 = /constructor/i.test(window.Element);

        // 推送通知 API 暴露在 window 对象上
        // 使用默认参数值以避免对 undefined 调用 toString()
        // Safari 7.1 及以上版本支持
        this.isSafari_Gte7_1 =
            (({pushNotification = {}} = {}) =>
              pushNotification.toString() == '[object SafariRemoteNotification]'
            )(window.safari);

        // 测试 addons 属性
        // Opera 20 及以上版本支持
        this.isOpera_Gte20 = !!window.opr && !!window.opr.addons;
    }

    isIE() { return this.isIE_Gte6Lte10 || this.isIE_Gte7Lte11; }
    isEdge() { return this.isEdge_Gte20 && !this.isIE(); }
    isFirefox() { return this.isFirefox_Gte1; }
    isChrome() { return this.isChrome_Gte1; }
    isSafari() { return this.isSafari_Gte3Lte9_1 || this.isSafari_Gte7_1; }
    isOpera() { return this.isOpera_Gte20; }
}
```

这个类暴露的通用浏览器检测方法使用了检测浏览器范围的能力测试。随着浏览器的变迁及发展，可以不断调整底层检测逻辑，但主要的 API 可以保持不变。

3. 能力检测的局限

通过检测一种或一组能力，并不总能确定使用的是哪种浏览器。以下“浏览器检测”代码（或其他类似代码）经常出现在很多网站中，但都没有正确使用能力检测：

```
// 不要这样做！不够特殊
let isFirefox = !!(navigator.vendor && navigator.vendorSub);

// 不要这样做！假设太多
let isIE = !!(document.all && document.uniqueID);
```

这是错误使用能力检测的典型示例。过去，Firefox 可以通过 `navigator.vendor` 和 `navigator.vendorSub` 来检测，但后来 `Safari` 也实现了同样的属性，于是这段代码就会产生误报。为确定 IE，这段代码检测了 `document.all` 和 `document.uniqueID`。这是假设 IE 将来的版本中还会继续存在这两个属性，而且其他浏览器也不会实现它们。不过这两个检测都使用双重否定操作符来产生布尔值（这