

第 25 章

新兴的 API

本章内容

- 创建平滑的动画
- 操作文件
- 使用 Web Workers 在后台执行 JavaScript

随着 HTML5 的出现，面向未来 Web 应用的 JavaScript API 也得到了极大的发展。这些 API 没有包含在 HTML5 规范中，而是各自有各自的规范。但是，它们都属于“HTML5 相关的 API”。本章介绍的所有 API 都在持续制定中，还没有完全固定下来。

无论如何，浏览器已经着手实现这些 API，而 Web 应用开发人员也都开始使用它们了。读者应该能够注意到，其中很多 API 都带有特定于浏览器的前缀，比如微软是 ms，而 Chrome 和 Safari 是 webkit。通过添加这些前缀，不同的浏览器可以测试还在开发中的新 API，不过请记住，去掉前缀之后的部分在所有浏览器中都是一致的。

25.1 requestAnimationFrame()

很长时间以来，计时器和循环间隔一直都是 JavaScript 动画的最核心技术。虽然 CSS 变换及动画为 Web 开发人员提供了实现动画的简单手段，但 JavaScript 动画开发领域的状况这些年来并没有大的变化。Firefox 4 最早为 JavaScript 动画添加了一个新 API，即 `mozRequestAnimationFrame()`。这个方法会告诉浏览器：有一个动画开始了。进而浏览器就可以确定重绘的最佳方式。

25.1.1 早期动画循环

在 JavaScript 中创建动画的典型方式，就是使用 `setInterval()` 方法来控制所有动画。以下是一个使用 `setInterval()` 的基本动画循环：

```
(function(){
    function updateAnimations(){
        doAnimation1();
        doAnimation2();
        //其他动画
    }

    setInterval(updateAnimations, 100);
})();
```

为了创建一个小型动画库，`updateAnimations()`方法就得不断循环地运行每个动画，并相应地改变不同元素的状态（例如，同时显示一个新闻跑马灯和一个进度条）。如果没有动画需要更新，这个方法可以退出，什么也不用做，甚至可以把动画循环停下来，等待下一次需要更新的动画。

编写这种动画循环的关键是要知道延迟时间多长合适。一方面，循环间隔必须足够短，这样才能让不同的动画效果显得更平滑流畅；另一方面，循环间隔还要足够长，这样才能确保浏览器有能力渲染产生的变化。大多数电脑显示器的刷新频率是 60Hz，大概相当于每秒钟重绘 60 次。大多数浏览器都会对重绘操作加以限制，不超过显示器的重绘频率，因为即使超过那个频率用户体验也不会有提升。

因此，最平滑动画的最佳循环间隔是 1000ms/60，约等于 17ms。以这个循环间隔重绘的动画是最平滑的，因为这个速度最接近浏览器的最高限速。为了适应 17ms 的循环间隔，多重动画可能需要加以节制，以便不会完成得太快。

虽然与使用多组 `setTimeout()` 的循环方式相比，使用 `setInterval()` 的动画循环效率更高，但后者也不是没有问题。无论是 `setInterval()` 还是 `setTimeout()` 都不十分精确。为它们传入的第二个参数，实际上只是指定了把动画代码添加到浏览器 UI 线程队列中以等待执行的时间。如果队列前面已经加入了其他任务，那动画代码就要等前面的任务完成后再执行。简言之，以毫秒表示的延迟时间并不代表到时候一定会执行动画代码，而仅代表到时候会把代码添加到任务队列中。如果 UI 线程繁忙，比如忙于处理用户操作，那么即使把代码加入队列也不会立即执行。

25.1.2 循环间隔的问题

知道什么时候绘制下一帧是保证动画平滑的关键。然而，直至最近，开发人员都没有办法确保浏览器按时绘制下一帧。随着<canvas>元素越来越流行，新的基于浏览器的游戏也开始崭露头角，面对不十分精确的 `setInterval()` 和 `setTimeout()`，开发人员一筹莫展。

浏览器使用的计时器的精度进一步恶化了问题。具体地说，浏览器使用的计时器并非精确到毫秒级别。以下是几个浏览器的计时器精度。

- ❑ IE8 及更早版本的计时器精度为 15.625ms。
- ❑ IE9 及更晚版本的计时器精度为 4ms。
- ❑ Firefox 和 Safari 的计时器精度大约为 10ms。
- ❑ Chrome 的计时器精度为 4ms。

IE9 之前版本的计时器精度为 15.625ms，因此介于 0 和 15 之间的任何值只能是 0 和 15。IE9 把计时器精度提高到了 4ms，但这个精度对于动画来说仍然不够明确。Chrome 的计时器精度为 4ms，而 Firefox 和 Safari 的精度是 10ms。更为复杂的是，浏览器都开始限制后台标签页或不活动标签页的计时器。因此，即使你优化了循环间隔，结果仍然只能接近你想要的效果。

25.1.3 `mozRequestAnimationFrame`

Mozilla 的 Robert O’Callahan 认识到了这个问题，提出了一个非常独特的方案。他指出，CSS 变换和动画的优势在于浏览器知道动画什么时候开始，因此会计算出正确的循环间隔，在恰当的时候刷新 UI。而对于 JavaScript 动画，浏览器无从知晓什么时候开始。因此他的方案就是创造一个新方法 `mozRequestAnimationFrame()`，通过它告诉浏览器某些 JavaScript 代码将要执行动画。这样浏览器可以在运行某些代码后进行适当的优化。

`mozRequestAnimationFrame()`方法接收一个参数,即在重绘屏幕前调用的一个函数。这个函数负责改变下一次重绘时的 DOM 样式。为了创建动画循环,可以像以前使用 `setTimeout()` 一样,把多个对 `mozRequestAnimationFrame()` 的调用连缀起来。比如:

```
function updateProgress(){
    var div = document.getElementById("status");
    div.style.width = (parseInt(div.style.width, 10) + 5) + "%";

    if (div.style.left != "100%"){
        mozRequestAnimationFrame(updateProgress);
    }
}

mozRequestAnimationFrame(updateProgress);
```

因为 `mozRequestAnimationFrame()` 只运行一次传入的函数,因此在需要再次修改 UI 从而生成动画时,需要再次手工调用它。同样,也需要同时考虑什么时候停止动画。这样就能得到非常平滑流畅的动画。

目前来看, `mozRequestAnimationFrame()` 解决了浏览器不知道 JavaScript 动画什么时候开始、不知道最佳循环间隔时间的问题,但不知道代码到底什么时候执行的问题呢? 同样的方案也可以解决这个问题。

我们传递的 `mozRequestAnimationFrame()` 函数也会接收一个参数,它是一个时间码(从1970年1月1日起至今的毫秒数),表示下一次重绘的实际发生时间。注意,这一点很重要: `mozRequestAnimationFrame()` 会根据这个时间码设定将来的某个时刻进行重绘,而根据这个时间码,你也能知道那个时刻是什么时间。然后,再优化动画效果就有了依据。

要知道距离上一次重绘已经过去了多长时间,可以查询 `mozAnimationStartTime`,其中包含上一次重绘的时间码。用传入回调函数的时间码减去这个时间码,就能计算出在屏幕上重绘下一组变化之前要经过多长时间。使用这个值的典型方式如下:

```
function draw(timestamp){

    //计算两次重绘的时间间隔
    var diff = timestamp - startTime;

    //使用 diff 确定下一步的绘制时间

    //把 startTime 重写为这一次的绘制时间
    startTime = timestamp;

    //重绘 UI
    mozRequestAnimationFrame(draw);
}

var startTime = mozAnimationStartTime;
mozRequestAnimationFrame(draw);
```

这里的关键是第一次读取 `mozAnimationStartTime` 的值,必须在传递给 `mozRequestAnimationFrame()` 的回调函数外面进行。如果是在回调函数内部读取 `mozAnimationStartTime`,得到的值与传入的时间码是相等的。

25.1.4 webkitRequestAnimationFrame 与 msRequestAnimationFrame

基于 `mozRequestAnimationFrame()`, **Chrome** 和 **IE10+**也都给出了自己的实现, 分别叫 `webkitRequestAnimationFrame()` 和 `msRequestAnimationFrame()`。这两个版本与 **Mozilla** 的版本有两个方面的微小差异。首先, 不会给回调函数传递时间码, 因此你无法知道下一次重绘将发生在什么时间。其次, **Chrome** 又增加了第二个可选的参数, 即将要发生变化的 **DOM** 元素。知道了重绘将发生在页面中哪个特定元素的区域内, 就可以将重绘限定在该区域中。

既然没有下一次重绘的时间码, 那 **Chrome** 和 **IE** 没有提供 `mozAnimationStartTime` 的实现也就很容易理解了——没有那个时间码, 实现这个属性也没有什么用。不过, **Chrome** 倒是又提供了另一个方法 `webkitCancelAnimationFrame()`, 用于取消之前计划执行的重绘操作。

假如你不需要知道精确的时间差, 那么可以在 **Firefox 4+**、**IE10+**和 **Chrome** 中可以参考以下模式创建动画循环。

```
(function(){  
  
    function draw(timestamp){  
  
        //计算两次重绘的时间间隔  
        var drawStart = {timestamp || Date.now()},  
            diff = drawStart - startTime;  
  
        //使用 diff 确定下一步的绘制时间  
  
        //把 startTime 重写为这一次的绘制时间  
        startTime = drawStart;  
  
        //重绘 UI  
        requestAnimationFrame(draw);  
    }  
  
    var requestAnimationFrame = window.requestAnimationFrame ||  
        window.mozRequestAnimationFrame ||  
        window.webkitRequestAnimationFrame ||  
        window.msRequestAnimationFrame,  
        startTime = window.mozAnimationStartTime || Date.now();  
    requestAnimationFrame(draw);  
})();
```

以上模式利用已有的功能创建了一个动画循环, 大致计算出了两次重绘的时间间隔。在 **Firefox** 中, 计算时间间隔使用的是既有的时间码, 而在 **Chrome** 和 **IE** 中, 则使用不十分精确的 `Date` 对象。这个模式可以大致体现出两次重绘的时间间隔, 但不会告诉你在 **Chrome** 和 **IE** 中的时间间隔到底是多少。不过, 大致知道时间间隔总比一点儿概念也没有好些。

因为首先检测的是标准函数名, 其次才是特定于浏览器的版本, 所以这个动画循环在将来也能够使用。

目前, **W3C** 已经着手起草 `requestAnimationFrame()` API, 而且作为 **Web Performance Group** 的一部分, **Mozilla** 和 **Google** 正共同参与该标准草案的制定工作。

25.2 Page Visibility API

不知道用户是不是正在与页面交互，这是困扰广大 Web 开发人员的一个主要问题。如果页面最小化了或者隐藏在了其他标签页后面，那么有些功能是可以停下来的，比如轮询服务器或者某些动画效果。而 Page Visibility API（页面可见性 API）就是为了让开发人员知道页面是否对用户可见而推出的。

这个 API 本身非常简单，由以下三部分组成。

- `document.hidden`: 表示页面是否隐藏的布尔值。页面隐藏包括页面在后台标签页中或者浏览器最小化。
- `document.visibilityState`: 表示下列 4 个可能状态的值。
 - 页面在后台标签页中或浏览器最小化。
 - 页面在前台标签页中。
 - 实际的页面已经隐藏，但用户可以看到页面的预览（就像在 Windows 7 中，用户把鼠标移动到任务栏的图标上，就可以显示浏览器中当前页面的预览）。
 - 页面在屏幕外执行预渲染处理。
- `visibilitychange` 事件：当文档从可见变为不可见或从不可见变为可见时，触发该事件。

在编写本书时，只有 IE10 和 Chrome 支持 Page Visibility API。IE 的版本是在每个属性或事件前面加上 `ms` 前缀，而 Chrome 则是加上 `webkit` 前缀。因此 `document.hidden` 在 IE 的实现中就是 `document.msHidden`，而在 Chrome 的实现中则是 `document.webkitHidden`。检查浏览器是否支持这个 API 的最佳方式如下：

```
function isHiddenSupported(){  
    return typeof (document.hidden || document.msHidden ||  
        document.webkitHidden) != "undefined";  
}
```

PageVisibilityAPIExample01.htm

类似地，使用同样的模式可以检测页面是否隐藏：

```
if (document.hidden || document.msHidden || document.webKitHidden){  
    // 页面隐藏了  
} else {  
    // 页面未隐藏  
}
```

PageVisibilityAPIExample01.htm

注意，以上代码在不支持该 API 的浏览器中会提示页面未隐藏。这是 Page Visibility API 有意设计的结果，目的是为了向后兼容。

为了在页面从可见变为不可见或从不可见变为可见时收到通知，可以侦听 `visibilitychange` 事件。在 IE 中，这个事件叫 `msvisibilitychange`，而在 Chrome 中这个事件叫 `webkitvisibilitychange`。为了在两个浏览器中都能侦听到该事件，可以像下面的例子一样，为每个事件都指定相同的事件处理程序：

```
function handleVisibilityChange(){
    var output = document.getElementById("output"),
        msg;

    if (document.hidden || document.msHidden || document.webkitHidden){
        msg = "Page is now hidden. " + (new Date()) + "<br>";
    } else {
        msg = "Page is now visible. " + (new Date()) + "<br>";
    }

    output.innerHTML += msg;
}

//要为两个事件都指定事件处理程序
EventUtil.addHandler(document, "msvisibilitychange", handleVisibilityChange);
EventUtil.addHandler(document, "webkitvisibilitychange", handleVisibilityChange);
```

PageVisibilityAPIExample01.htm

以上代码同时适用于 IE 和 Chrome。而且，API 的这一部分已经相对稳定，因此在实际的 Web 开发中也可以使用以上代码。

关于这一 API 的实现，差异最大的是 `document.visibilityState` 属性。IE10 PR 2 的 `document.msVisibilityState` 是一个表示如下 4 种状态的数字值。

- (1) `document.MS_PAGE_HIDDEN` (0)
- (2) `document.MS_PAGE_VISIBLE` (1)
- (3) `document.MS_PAGE_PREVIEW` (2)
- (4) `document.MS_PAGE_PRERENDER` (3)

在 Chrome 中，`document.webkitVisibilityState` 可能是下列 3 个字符串值：

- (1) "hidden"
- (2) "visible"
- (3) "prerender"

Chrome 并没有给每个状态定义对应的常量，但最终的实现很可能会使用常量。

由于存在以上差异，所以建议大家先不要完全依赖带前缀的 `document.visibilityState`，最好只使用 `document.hidden` 属性。

25.3 Geolocation API

地理定位 (geolocation) 是最令人兴奋，而且得到了广泛支持的一个新 API。通过这套 API，JavaScript 代码能够访问到用户的当前位置信息。当然，访问之前必须得到用户的明确许可，即同意在页面中共享其位置信息。如果页面尝试访问地理定位信息，浏览器就会显示一个对话框，请求用户许可共享其位置信息。图 25-1 展示了 Chrome 中的这样一个对话框。



图 25-1

Geolocation API 在浏览器中的实现是 `navigator.geolocation` 对象，这个对象包含 3 个方法。第一个方法是 `getCurrentPosition()`，调用这个方法就会触发请求用户共享地理定位信息的对话框。

这个方法接收 3 个参数：成功回调函数、可选的失败回调函数和可选的选项对象。

其中，成功回调函数会接收到一个 `Position` 对象参数，该对象有两个属性：`coords` 和 `timestamp`。而 `coords` 对象中将包含下列与位置相关的信息。

- ❑ `latitude`：以十进制度数表示的纬度。
- ❑ `longitude`：以十进制度数表示的经度。
- ❑ `accuracy`：经、纬度坐标的精度，以米为单位。

有些浏览器还可能会在 `coords` 对象中提供如下属性。

- ❑ `altitude`：以米为单位的海拔高度，如果没有相关数据则值为 `null`。
- ❑ `altitudeAccuracy`：海拔高度的精度，以米为单位，数值越大越不精确。
- ❑ `heading`：指南针的方向， 0° 表示正北，值为 `NaN` 表示没有检测到数据。
- ❑ `speed`：速度，即每秒移动多少米，如果没有相关数据则值为 `null`。

在实际开发中，`latitude` 和 `longitude` 是大多数 Web 应用最常用到的属性。例如，以下代码将在地图上绘制用户的位置：

```
navigator.geolocation.getCurrentPosition(function(position){
    drawMapCenteredAt(position.coords.latitude, position.coords.longitude);
});
```

以上介绍的是成功回调函数。`getCurrentPosition()` 的第二个参数，即失败回调函数，在被调用的时候也会接收到一个参数。这个参数是一个对象，包含两个属性：`message` 和 `code`。其中，`message` 属性中保存着给人看的文本消息，解释为什么会出错，而 `code` 属性中保存着一个数值，表示错误的类型：用户拒绝共享（1）、位置无效（2）或者超时（3）。实际开发中，大多数 Web 应用只会将错误消息保存到日志文件中，而不一定会因此修改用户界面。例如：

```
navigator.geolocation.getCurrentPosition(function(position){
    drawMapCenteredAt(position.coords.latitude, position.coords.longitude);
}, function(error){
    console.log("Error code: " + error.code);
    console.log("Error message: " + error.message);
});
```

`getCurrentPosition()` 的第三个参数是一个选项对象，用于设定信息的类型。可以设置的选项有三个：`enableHighAccuracy` 是一个布尔值，表示必须尽可能使用最准确的位置信息；`timeout` 是以毫秒数表示的等待位置信息的最长时间；`maximumAge` 表示上一次取得的坐标信息的有效时间，以毫秒表示，如果时间到则重新取得新坐标信息。例如：

```
navigator.geolocation.getCurrentPosition(function(position){
    drawMapCenteredAt(position.coords.latitude, position.coords.longitude);
}, function(error){
    console.log("Error code: " + error.code);
    console.log("Error message: " + error.message);
}, {
    enableHighAccuracy: true,
    timeout: 5000,
    maximumAge: 25000
});
```

这三个选项都是可选的，可以单独设置，也可以与其他选项一起设置。除非确实需要非常精确的信息，否则建议保持 `enableHighAccuracy` 的 `false` 值（默认值）。将这个选项设置为 `true` 需要更长的时间，而且在移动设备上还会导致消耗更多电量。类似地，如果不需要频繁更新用户的位置信息，那么可以将 `maximumAge` 设置为 `Infinity`，从而始终都使用上一次的坐标信息。

如果你希望跟踪用户的位置，那么可以使用另一个方法 `watchPosition()`。这个方法接收的参数与 `getCurrentPosition()` 方法完全相同。实际上，`watchPosition()` 与定时调用 `getCurrentPosition()` 的效果相同。在第一次调用 `watchPosition()` 方法后，会取得当前位置，执行成功回调或者错误回调。然后，`watchPosition()` 就地等待系统发出位置已改变的信号（它不会自己轮询位置）。

调用 `watchPosition()` 会返回一个数值标识符，用于跟踪监控的操作。基于这个返回值可以取消监控操作，只要将其传递给 `clearWatch()` 方法即可（与使用 `setTimeout()` 和 `clearTimeout()` 类似）。例如：

```
var watchId = navigator.geolocation.watchPosition(function(position){
    drawMapCenteredAt(position.coords.latitude, position.coords.longitude);
}, function(error){
    console.log("Error code: " + error.code);
    console.log("Error message: " + error.message);
});

clearWatch(watchId);
```

以上例子调用了 `watchPosition()` 方法，将返回的标识符保存在了 `watchId` 中。然后，又将 `watchId` 传给了 `clearWatch()`，取消了监控操作。

支持地理定位的浏览器有 IE9+、Firefox 3.5+、Opera 10.6+、Safari 5+、Chrome、iOS 版 Safari、Android 版 WebKit。要了解使用地理定位的更多精彩范例，请访问 <http://html5demos.com/geo>。

25.4 File API

不能直接访问用户计算机中的文件，一直都是 Web 应用开发中的一大障碍。2000 年以前，处理文件的唯一方式就是在表单中加入 `<input type="file">` 字段，仅此而已。File API（文件 API）的宗旨是为 Web 开发人员提供一种安全的方式，以便在客户端访问用户计算机中的文件，并更好地对这些文件执行操作。支持 File API 的浏览器有 IE10+、Firefox 4+、Safari 5.0.5+、Opera 11.1+ 和 Chrome。

File API 在表单中的文件输入字段的基础上，又添加了一些直接访问文件信息的接口。HTML5 在 DOM 中为文件输入元素添加了一个 `files` 集合。在通过文件输入字段选择了一或多个文件时，`files` 集合中将包含一组 File 对象，每个 File 对象对应着一个文件。每个 File 对象都有下列只读属性。

- ❑ `name`：本地文件系统上的文件名。
- ❑ `size`：文件的字节大小。
- ❑ `type`：字符串，文件的 MIME 类型。
- ❑ `lastModifiedDate`：字符串，文件上一次被修改的时间（只有 Chrome 实现了这个属性）。

举个例子，通过侦听 `change` 事件并读取 `files` 集合就可以知道选择的每个文件的信息：

```
var fileList = document.getElementById("files-list");
EventUtil.addHandler(fileList, "change", function(event){
```




```
var files = EventUtil.getTarget(event).files,
    i = 0,
    len = files.length;

while (i < len){
    console.log(files[i].name + " (" + files[i].type + ", " + files[i].size +
        " bytes) ");
    i++;
}
});
```

FileAPIExample01.htm

这个例子把每个文件的信息输出到了控制台中。仅仅这一项功能，对 Web 应用开发来说就已经是非常大的进步了。不过，File API 的功能还不止于此，通过它提供的 `FileReader` 类型甚至还可以读取文件中的数据。

25.4.1 `FileReader` 类型

`FileReader` 类型实现的是一种异步文件读取机制。可以把 `FileReader` 想象成 `XMLHttpRequest`，区别只是它读取的是文件系统，而不是远程服务器。为了读取文件中的数据，`FileReader` 提供了如下几个方法。

- ❑ `readAsText(file, encoding)`：以纯文本形式读取文件，将读取到的文本保存在 `result` 属性中。第二个参数用于指定编码类型，是可选的。
- ❑ `readAsDataURL(file)`：读取文件并将文件以数据 URI 的形式保存在 `result` 属性中。
- ❑ `readAsBinaryString(file)`：读取文件并将一个字符串保存在 `result` 属性中，字符串中的每个字符表示一字节。
- ❑ `readAsArrayBuffer(file)`：读取文件并将一个包含文件内容的 `ArrayBuffer` 保存在 `result` 属性中。

这些读取文件的方法为灵活地处理文件数据提供了极大便利。例如，可以读取图像文件并将其保存为数据 URI，以便将其显示给用户，或者为了解析方便，可以将文件读取为文本形式。

由于读取过程是异步的，因此 `FileReader` 也提供了几个事件。其中最有用的三个事件是 `progress`、`error` 和 `load`，分别表示是否又读取了新数据、是否发生了错误以及是否已经读完了整个文件。

每过 50ms 左右，就会触发一次 `progress` 事件，通过事件对象可以获得与 XHR 的 `progress` 事件相同的信息（属性）：`lengthComputable`、`loaded` 和 `total`。另外，尽管可能没有包含全部数据，但每次 `progress` 事件中都可以通过 `FileReader` 的 `result` 属性读取到文件内容。

由于种种原因无法读取文件，就会触发 `error` 事件。触发 `error` 事件时，相关的信息将保存到 `FileReader` 的 `error` 属性中。这个属性中将保存一个对象，该对象只有一个属性 `code`，即错误码。这个错误码是 1 表示未找到文件，是 2 表示安全性错误，是 3 表示读取中断，是 4 表示文件不可读，是 5 表示编码错误。

文件成功加载后会触发 `load` 事件；如果发生了 `error` 事件，就不会发生 `load` 事件。以下是一个使用上述三个事件的例子。



```
var fileList = document.getElementById("files-list");
EventUtil.addHandler(fileList, "change", function(event){
    var info = "",
        output = document.getElementById("output"),
        progress = document.getElementById("progress"),
        files = EventUtil.getTarget(event).files,
        type = "default",
        reader = new FileReader();

    if (/image/.test(files[0].type)){
        reader.readAsDataURL(files[0]);
        type = "image";
    } else {
        reader.readAsText(files[0]);
        type = "text";
    }

    reader.onerror = function(){
        output.innerHTML = "Could not read file, error code is " +
            reader.error.code;
    };

    reader.onprogress = function(event){
        if (event.lengthComputable){
            progress.innerHTML = event.loaded + "/" + event.total;
        }
    };

    reader.onload = function(){
        var html = "";
        switch(type){
            case "image":
                html = "<img src=\"\" + reader.result + \"\">";
                break;
            case "text":
                html = reader.result;
                break;
        }
        output.innerHTML = html;
    };
});
```

FileAPIExample02.htm

这个例子读取了表单字段中选择的文件，并将其内容显示在了页面中。如果文件有 MIMI 类型，表示文件是图像，因此在 load 事件中就把它保存为数据 URI，并在页面中将这幅图像显示出来。如果文件不是图像，则以字符串形式读取文件内容，然后如实在页面中显示读取到的内容。这里使用了 progress 事件来跟踪读取了多少字节的数据，而 error 事件则用于监控发生的错误。

如果想中断读取过程，可以调用 abort() 方法，这样就会触发 abort 事件。在触发 load、error 或 abort 事件后，会触发另一个事件 loadend。loadend 事件发生就意味着已经读取完整个文件，或者读取时发生了错误，或者读取过程被中断。

实现 File API 的所有浏览器都支持 readAsText() 和 readAsDataURL() 方法。但 IE10 PR 2 并未实现 readAsBinaryString() 和 readAsArrayBuffer() 方法。

25.4.2 读取部分内容

有时候，我们只想读取文件的一部分而不是全部内容。为此，File 对象还支持一个 slice() 方法，这个方法在 Firefox 中的实现叫 mozSlice()，在 Chrome 中的实现叫 webkitSlice()，Safari 的 5.1 及之前版本不支持这个方法。slice() 方法接收两个参数：起始字节及要读取的字节数。这个方法返回一个 Blob 的实例，Blob 是 File 类型的父类型。下面是一个通用的函数，可以在不同实现中使用 slice() 方法：

```
function blobSlice(blob, startByte, length){
  if (blob.slice){
    return blob.slice(startByte, length);
  } else if (blob.webkitSlice){
    return blob.webkitSlice(startByte, length);
  } else if (blob.mozSlice){
    return blob.mozSlice(startByte, length);
  } else {
    return null;
  }
}
```

FileAPIExample03.htm

Blob 类型有一个 size 属性和一个 type 属性，而且它也支持 slice() 方法，以便进一步切割数据。通过 FileReader 也可以从 Blob 中读取数据。下面这个例子只读取文件的 32B 内容。

```
var fileList = document.getElementById("files-list");
EventUtil.addHandler(fileList, "change", function(event){
  var info = "",
      output = document.getElementById("output"),
      progress = document.getElementById("progress"),
      files = EventUtil.getTarget(event).files,
      reader = new FileReader(),
      blob = blobSlice(files[0], 0, 32);

  if (blob){
    reader.readAsText(blob);

    reader.onerror = function(){
      output.innerHTML = "Could not read file, error code is " +
        reader.error.code;
    };

    reader.onload = function(){
      output.innerHTML = reader.result;
    };
  } else {
    alert("Your browser doesn't support slice().");
  }
});
```

FileAPIExample03.htm

只读取文件的一部分可以节省时间，非常适合只关注数据中某个特定部分（如文件头部）的情况。

25.4.3 对象 URL

对象 URL 也被称为 blob URL，指的是引用保存在 File 或 Blob 中数据的 URL。使用对象 URL 的好处是可以不必把文件内容读取到 JavaScript 中而直接使用文件内容。为此，只要在需要文件内容的地方提供对象 URL 即可。要创建对象 URL，可以使用 `window.URL.createObjectURL()` 方法，并传入 File 或 Blob 对象。这个方法在 Chrome 中的实现叫 `window.webkitURL.createObjectURL()`，因此可以通过如下函数来消除命名的差异：

```
function createObjectURL(blob){
  if (window.URL){
    return window.URL.createObjectURL(blob);
  } else if (window.webkitURL){
    return window.webkitURL.createObjectURL(blob);
  } else {
    return null;
  }
}
```

FileAPIExample04.htm

这个函数的返回值是一个字符串，指向一块内存的地址。因为这个字符串是 URL，所以在 DOM 中也能使用。例如，以下代码可以在页面中显示一个图像文件：



```
var fileList = document.getElementById("files-list");
EventUtil.addHandler(fileList, "change", function(event){
  var info = "",
      output = document.getElementById("output"),
      progress = document.getElementById("progress"),
      files = EventUtil.getTarget(event).files,
      reader = new FileReader(),
      url = createObjectURL(files[0]);

  if (url){
    if (/image/.test(files[0].type)){
      output.innerHTML = "<img src=\"" + url + "\">";
    } else {
      output.innerHTML = "Not an image.";
    }
  } else {
    output.innerHTML = "Your browser doesn't support object URLs.";
  }
});
```

FileAPIExample04.htm

直接把对象 URL 放在 `` 标签中，就省去了把数据先读到 JavaScript 中的麻烦。另一方面，`` 标签则会找到相应的内存地址，直接读取数据并将图像显示在页面中。

如果不再需要相应的数据，最好释放它占用的内容。但只要有代码在引用对象 URL，内存就不会释放。要手工释放内存，可以把对象 URL 传给 `window.URL.revokeObjectURL()`（在 Chrome 中是 `window.webkitURL.revokeObjectURL()`）。要兼容这两种方法的实现，可以使用以下函数：

```
function revokeObjectURL(url){
    if (window.URL){
        window.URL.revokeObjectURL(url);
    } else if (window.webkitURL){
        window.webkitURL.revokeObjectURL(url);
    }
}
```

页面卸载时会自动释放对象 URL 占用的内存。不过，为了确保尽可能少地占用内存，最好在不需要某个对象 URL 时，就马上手工释放其占用的内存。

支持对象 URL 的浏览器有 IE10+、Firefox 4 和 Chrome。

25.4.4 读取拖放的文件

围绕读取文件信息，结合使用 HTML5 拖放 API 和文件 API，能够创造出令人瞩目的用户界面：在页面上创建了自定义的放置目标之后，你可以从桌面上把文件拖放到该目标。与拖放一张图片或者一个链接类似，从桌面上把文件拖放到浏览器中也会触发 drop 事件。而且可以在 event.dataTransfer.files 中读取到被放置的文件，当然此时它是一个 File 对象，与通过文件输入字段取得的 File 对象一样。

下面这个例子会将放置到页面中自定义的放置目标中的文件信息显示出来：

```
var droptarget = document.getElementById( "droptarget" );

function handleEvent(event){
    var info = "",
        output = document.getElementById("output"),
        files, i, len;

    EventUtil.preventDefault(event);

    if (event.type == "drop"){
        files = event.dataTransfer.files;
        i = 0;
        len = files.length;

        while (i < len){
            info += files[i].name + " (" + files[i].type + ", " + files[i].size +
                " bytes)<br>";
            i++;
        }

        output.innerHTML = info;
    }

    EventUtil.addHandler(droptarget, "dragenter", handleEvent);
    EventUtil.addHandler(droptarget, "dragover", handleEvent);
    EventUtil.addHandler(droptarget, "drop", handleEvent);
}
```

FileAPIExample05.htm

与之前展示的拖放示例一样，这里也必须取消 dragenter、dragover 和 drop 的默认行为。在 drop 事件中，可以通过 event.dataTransfer.files 读取文件信息。还有一种利用这个功能的流行做法，即结合 XMLHttpRequest 和拖放文件来实现上传。

25.4.5 使用 XHR 上传文件

通过 File API 能够访问到文件内容，利用这一点就可以通过 XHR 直接把文件上传到服务器。当然啦，把文件内容放到 `send()` 方法中，再通过 POST 请求，的确很容易就能实现上传。但这样做传递的是文件内容，因而服务器端必须收集提交的内容，然后再把它们保存到另一个文件中。其实，更好的做法是以表单提交的方式来上传文件。

这样使用 `FormData` 类型就很容易做到了(第 21 章介绍过 `FormData`)。首先，要创建一个 `FormData` 对象，通过它调用 `append()` 方法并传入相应的 `File` 对象作为参数。然后，再把 `FormData` 对象传递给 XHR 的 `send()` 方法，结果与通过表单上传一模一样。



```
var droptarget = document.getElementById("droptarget");

function handleEvent(event){
    var info = "",
        output = document.getElementById("output"),
        data, xhr,
        files, i, len;

    EventUtil.preventDefault(event);

    if (event.type == "drop"){
        data = new FormData();
        files = event.dataTransfer.files;
        i = 0;
        len = files.length;

        while (i < len){
            data.append("file" + i, files[i]);
            i++;
        }

        xhr = new XMLHttpRequest();
        xhr.open("post", "FileAPIExample06Upload.php", true);
        xhr.onreadystatechange = function(){
            if (xhr.readyState == 4){
                alert(xhr.responseText);
            }
        };
        xhr.send(data);
    }
}

EventUtil.addHandler(droptarget, "dragenter", handleEvent);
EventUtil.addHandler(droptarget, "dragover", handleEvent);
EventUtil.addHandler(droptarget, "drop", handleEvent);
```

FileAPIExample06.htm

这个例子创建一个 `FormData` 对象，与每个文件对应的键分别是 `file0`、`file1`、`file2` 这样的格式。注意，不用额外写任何代码，这些文件就可以作为表单的值提交。而且，也不必使用 `FileReader`，只要传入 `File` 对象即可。

使用 FormData 上传文件，在服务器端就好像是接收到了常规的表单数据一样，一切按部就班地处理即可。换句话说，如果服务器端使用的是 PHP，那么\$_FILES 数组中就会保存着上传的文件。支持以这种方式上传文件的浏览器有 Firefox 4+、Safari 5+和 Chrome。

25.5 Web 计时

页面性能一直都是 Web 开发人员最关注的领域。但直到最近，度量页面性能指标的唯一方式，就是提高代码复杂度程度和巧妙地使用 JavaScript 的 Date 对象。Web Timing API 改变了这个局面，让开发人员通过 JavaScript 就能使用浏览器内部的度量结果，通过直接读取这些信息可以做任何想要的分析。与本章介绍过的其他 API 不同，Web Timing API 实际上已经成为了 W3C 的建议标准，只不过目前支持它的浏览器还不够多。

Web 计时机制的核心是 window.performance 对象。对页面的所有度量信息，包括那些规范中已经定义的和将来才能确定的，都包含在这个对象里面。Web Timing 规范一开始就为 performance 对象定义了两个属性。

其中，performance.navigation 属性也是一个对象，包含着与页面导航有关的多个属性，如下所示。

- redirectCount: 页面加载前的重定向次数。
- type: 数值常量，表示刚刚发生的导航类型。
 - performance.navigation.TYPE_NAVIGATE (0): 页面第一次加载。
 - performance.navigation.TYPE_RELOAD (1): 页面重载过。
 - performance.navigation.TYPE_BACK_FORWARD (2): 页面是通过“后退”或“前进”按钮打开的。

另外，performance.timing 属性也是一个对象，但这个对象的属性都是时间戳（从软件纪元开始经过的毫秒数），不同的事件会产生不同的时间值。这些属性如下所示。

- navigationStart: 开始导航到当前页面的时间。
- unloadEventStart: 前一个页面的 unload 事件开始的时间。但只有在前一个页面与当前页面来自同一个域时这个属性才会有值；否则，值为 0。
- unloadEventEnd: 前一个页面的 unload 事件结束的时间。但只有在前一个页面与当前页面来自同一个域时这个属性才会有值；否则，值为 0。
- redirectStart: 到当前页面的重定向开始的时间。但只有在重定向的页面来自同一个域时这个属性才会有值；否则，值为 0。
- redirectEnd: 到当前页面的重定向结束的时间。但只有在重定向的页面来自同一个域时这个属性才会有值；否则，值为 0。
- fetchStart: 开始通过 HTTP GET 取得页面的时间。
- domainLookupStart: 开始查询当前页面 DNS 的时间。
- domainLookupEnd: 查询当前页面 DNS 结束的时间。
- connectStart: 浏览器尝试连接服务器的时间。
- connectEnd: 浏览器成功连接到服务器的时间。
- secureConnectionStart: 浏览器尝试以 SSL 方式连接服务器的时间。不使用 SSL 方式连接时，这个属性的值为 0。

- ❑ requestStart: 浏览器开始请求页面的时间。
- ❑ responseStart: 浏览器接收到页面第一字节的时间。
- ❑ responseEnd: 浏览器接收到页面所有内容的时间。
- ❑ domLoading: document.readyState 变为"loading"的时间。
- ❑ domInteractive: document.readyState 变为"interactive"的时间。
- ❑ domContentLoadedEventStart: 发生 DOMContentLoaded 事件的时间。
- ❑ domContentLoadedEventEnd: DOMContentLoaded 事件已经发生且执行完所有事件处理程序的时间。
- ❑ domComplete: document.readyState 变为"complete"的时间。
- ❑ loadEventStart: 发生 load 事件的时间。
- ❑ loadEventEnd: load 事件已经发生且执行完所有事件处理程序的时间。

通过这些时间值,就可以全面了解页面在被加载到浏览器的过程中都经历了哪些阶段,而哪些阶段可能是影响性能的瓶颈。给大家推荐一个使用 Web Timing API 的绝好示例,地址是 <http://webtimingdemo.appspot.com/>。

支持 Web Timing API 的浏览器有 IE10+和 Chrome。

25.6 Web Workers

随着 Web 应用复杂性的与日俱增,越来越复杂的计算在所难免。长时间运行的 JavaScript 进程会导致浏览器冻结用户界面,让人感觉屏幕“冻结”了。Web Workers 规范通过让 JavaScript 在后台运行解决了这个问题。浏览器实现 Web Workers 规范的方式有很多种,可以使用线程、后台进程或者运行在其他处理器核心上的进程,等等。具体的实现细节其实没有那么重要,重要的是开发人员现在可以放心地运行 JavaScript,而不必担心会影响用户体验了。

目前支持 Web Workers 的浏览器有 IE10+、Firefox 3.5+、Safari 4+、Opera 10.6+、Chrome 和 iOS 版的 Safari。

25.6.1 使用 Worker

实例化 Worker 对象并传入要执行的 JavaScript 文件名就可以创建一个新的 Web Worker。例如:

```
var worker = new Worker("stufftodo.js");
```

这行代码会导致浏览器下载 stufftodo.js,但只有 Worker 接收到消息才会实际执行文件中的代码。要给 Worker 传递消息,可以使用 postMessage() 方法(与 XDM 中的 postMessage() 方法类似):

```
worker.postMessage("start!");
```

消息内容可以是任何能够被序列化的值,不过与 XDM 不同的是,在所有支持的浏览器中,postMessage() 都能接收对象参数(Safari 4 是支持 Web Workers 的浏览器中最后一个只支持字符串参数的)。因此,可以随便传递任何形式的对象数据,如下面的例子所示:

```
worker.postMessage({
  type: "command",
  message: "start!"
});
```


一般来说，可以序列化为 JSON 结构的任何值都可以作为参数传递给 `postMessage()`。换句话说，这就意味着传入的值是被复制到 Worker 中，而非直接传过去的（与 XDM 类似）。

Worker 是通过 `message` 和 `error` 事件与页面通信的。这里的 `message` 事件与 XDM 中的 `message` 事件行为相同，来自 Worker 的数据保存在 `event.data` 中。Worker 返回的数据也可以是任何能够被序列化的值：

```
worker.onmessage = function(event){
    var data = event.data;

    //对数据进行处理
}
```

Worker 不能完成给定的任务时会触发 `error` 事件。具体来说，Worker 内部的 JavaScript 在执行过程中只要遇到错误，就会触发 `error` 事件。发生 `error` 事件时，事件对象中包含三个属性：`filename`、`lineno` 和 `message`，分别表示发生错误的文件名、代码行号和完整的错误消息。

```
worker.onerror = function(event){
    console.log("ERROR: " + event.filename + " (" + event.lineno + "): " +
        event.message);
};
```

建议大家在使用 Web Workers 时，始终都要使用 `onerror` 事件处理程序，即使这个函数（像上面例子所示的）除了把错误记录到日志中什么也不做都可以。否则，Worker 就会在发生错误时，悄无声息地失败了。

任何时候，只要调用 `terminate()` 方法就可以停止 Worker 的工作。而且，Worker 中的代码会立即停止执行，后续的所有过程都不会再发生（包括 `error` 和 `message` 事件也不会再触发）。

```
worker.terminate();    //立即停止 Worker 的工作
```

25.6.2 Worker 全局作用域

关于 Web Worker，最重要的是要知道它所执行的 JavaScript 代码完全在另一个作用域中，与当前网页中的代码不共享作用域。在 Web Worker 中，同样有一个全局对象和其他对象以及方法。但是，Web Worker 中的代码不能访问 DOM，也无法通过任何方式影响页面的外观。

Web Worker 中的全局对象是 `worker` 对象本身。也就是说，在这个特殊的全局作用域中，`this` 和 `self` 引用的都是 `worker` 对象。为便于处理数据，Web Worker 本身也是一个最小化的运行环境。

- ❑ 最小化的 `navigator` 对象，包括 `onLine`、`appName`、`appVersion`、`userAgent` 和 `platform` 属性；
- ❑ 只读的 `location` 对象；
- ❑ `setTimeout()`、`setInterval()`、`clearTimeout()` 和 `clearInterval()` 方法；
- ❑ `XMLHttpRequest` 构造函数。


显然，Web Worker 的运行环境与页面环境相比，功能是相当有限的。

当页面在 `worker` 对象上调用 `postMessage()` 时，数据会以异步方式被传递给 `worker`，进而触发 `worker` 中的 `message` 事件。为了处理来自页面的数据，同样也需要创建一个 `onmessage` 事件处理程序。

```
//Web Worker 内部的代码
self.onmessage = function(event){
    var data = event.data;

    //处理数据
};
```

大家看清楚，这里的 `self` 引用的是 `Worker` 全局作用域中的 `worker` 对象（与页面中的 `Worker` 对象不同一个对象）。`Worker` 完成工作后，通过调用 `postMessage()` 可以把数据再发回页面。例如，下面的例子假设需要 `Worker` 对传入的数组进行排序，而 `Worker` 在排序之后又将数组发回了页面：



```
//Web Worker 内部的代码
self.onmessage = function(event){
    var data = event.data;

    //别忘了，默认的 sort() 方法只比较字符串
    data.sort(function(a, b){
        return a - b;
    });
    self.postMessage(data);
};
```

WebWorkerExample01.js

传递消息就是页面与 `Worker` 相互之间通信的方式。在 `Worker` 中调用 `postMessage()` 会以异步方式触发页面中 `Worker` 实例的 `message` 事件。如果页面想要使用这个 `Worker`，可以这样：

```
//在页面中
var data = [23,4,7,9,2,14,6,651,87,41,7798,24],
    worker = new Worker("WebWorkerExample01.js");

worker.onmessage = function(event){
    var data = event.data;

    //对排序后的数组进行操作
};

//将数组发送给 worker 排序
worker.postMessage(data);
```

WebWorkerExample01.htm

排序的确是比较消耗时间的操作，因此转交给 `Worker` 做就不会阻塞用户界面了。另外，把彩色图像转换成灰阶图像以及加密解密之类的操作也是相当费时的。

在 `Worker` 内部，调用 `close()` 方法也可以停止工作。就像在页面中调用 `terminate()` 方法一样，`Worker` 停止工作后就不会再有事件发生了。

```
//Web Worker 内部的代码
self.close();
```

25.6.3 包含其他脚本

既然无法在 `Worker` 中动态创建新的 `<script>` 元素，那是不是就不能向 `Worker` 中添加其他脚本了

呢？不是，Worker 的全局作用域提供这个功能，即我们可以调用 `importScripts()` 方法。这个方法接收一个或多个指向 JavaScript 文件的 URL。每个加载过程都是异步进行的，因此所有脚本加载并执行之后，`importScripts()` 才会执行。例如：

```
//Web Worker 内部的代码
importScripts("file1.js", "file2.js");
```

即使 `file2.js` 先于 `file1.js` 下载完，执行的时候仍然会按照先后顺序执行。而且，这些脚本是在 Worker 的全局作用域中执行，如果脚本中包含与页面有关的 JavaScript 代码，那么脚本可能无法正确运行。请记住，Worker 中的脚本一般都具有特殊的用途，不会像页面中的脚本那么功能宽泛。

25.6.4 Web Workers 的未来

Web Workers 规范还在继续制定和改进之中。本节所讨论的 Worker 目前被称为“专用 Worker”（dedicated worker），因为它们是专门为某个特定的页面服务的，不能在页面间共享。该规范的另外一个概念是“共享 Worker”（shared worker），这种 Worker 可以在浏览器的多个标签中打开的同一个页面间共享。虽然 Safari 5、Chrome 和 Opera 10.6 都实现了共享 Worker，但由于该规范尚未完稿，因此很可能还会有变动。

另外，关于在 Worker 内部能访问什么不能访问什么，到如今仍然争论不休。有人认为 Worker 应该像页面一样能够访问任意数据，不光是 XHR，还有 `localStorage`、`sessionStorage`、`Indexed DB`、`Web Sockets`、`Server-Send Events` 等。好像支持这个观点的人更多一些，因此未来的 Worker 全局作用域很可能会有更大的空间。

25.7 小结

与 HTML5 同时兴起的是另外一批 JavaScript API。从技术规范角度讲，这批 API 不属于 HTML5，但从整体上可以称它们为 HTML5 JavaScript API。这些 API 的标准有不少虽然还在制定当中，但已经得到了浏览器的广泛支持，因此本章重点讨论了它们。

- ❑ `requestAnimationFrame()`：是一个着眼于优化 JavaScript 动画的 API，能够在动画运行期间发出信号。通过这种机制，浏览器就能够自动优化屏幕重绘操作。
- ❑ Page Visibility API：让开发人员知道用户什么时候正在看着页面，而什么时候页面是隐藏的。
- ❑ Geolocation API：在得到许可的情况下，可以确定用户所在的位置。在移动 Web 应用中，这个 API 非常重要而且常用。
- ❑ File API：可以读取文件内容，用于显示、处理和上传。与 HTML5 的拖放功能结合，很容易就能创造出拖放上传功能。
- ❑ Web Timing：给出了页面加载和渲染过程的很多信息，对性能优化非常有价值。
- ❑ Web Workers：可以运行异步 JavaScript 代码，避免阻塞用户界面。在执行复杂计算和数据处理的时候，这个 API 非常有用；要不然，这些任务轻则会占用很长时间，重则会导致用户无法与页面交互。